Development of a Scientific Visualization System

Thesis submitted in fulfillment of the requirements for the award of the degree of Doctor in de ingenieurswetenschappen (Doctor in Engineering) by

Dean Vučinić

July 2007

Advisor(s): Prof. Dr. Ir. Chris Lacor
             Prof. Dr. Ir. Charles Hirsch
CFView - Computational Flow Field Visualization System
to my beloved wife and children
Ester, Petra and Stefan

and to the memory of my mother Alice
PhD Committee

President:
Prof. Dr. Jacques Tiberghien
Vrije Universiteit Brussel (VUB)
Department of Electronics and Informatics (ETRO)

Vice President:
Prof. Dr. Ir. Rik Pintelon
Vrije Universiteit Brussel (VUB)
Department of Fundamental Electricity and Instrumentation (ELEC)

Secretary:
Prof. Dr. Ir. Jacques De Ruyck
Vrije Universiteit Brussel (VUB)
Head of the Mechanical Engineering Department (MECH)

Advisors:

Prof. Dr. Ir. Chris Lacor
Vrije Universiteit Brussel (VUB)
Department of Mechanical Engineering (MECH)
Head of the Fluid Mechanics and Thermodynamics Research Group

Prof. em. Dr. Ir. Charles Hirsch
Vrije Universiteit Brussel (VUB)
President of NUMECA International

Members:

Prof. Dr. Ir. Jan Cornelis
Vrije Universiteit Brussel (VUB)
Vice-rector for Research and Development
Head of the Electronics and Informatics department (ETRO)

Prof. Dr. Ir. Herman Deconinck
von Karman Institute for Fluid Mechanics (VKI)
Head of Aeronautics and Aerospace Department

Dr. François-Xavier Josset
Thales Research & Technology (TRT) France
Department of Cognitive Solutions

Prof. Dr. Arthur Rizzi
Royal Institute of Technology (KTH) Sweden
Head of Aerodynamics Division

Private defense held in Brussels on July 2, 2007 at 15:00
Public defense to be held in Brussels on September 5, 2007 at 17:00
TABLE OF CONTENTS

ABSTRACT ......................................................................................................................... III
ACKNOWLEDGMENTS ........................................................................................................ IV
NOMENCLATURE ................................................................................................................ VI
LIST OF FIGURES ............................................................................................................... VIII
LIST OF TABLES ................................................................................................................ XII

Introduction ......................................................................................................................... 1

1.1 SCIENTIFIC VISUALIZATION MODEL ..................................................................... 2

1.2 SCIENTIFIC VISUALIZATION ENVIRONMENT ......................................................... 4

1.3 SCIENTIFIC VISUALIZATION SOFTWARE - STATE OF THE ART ......................... 6

1.4 OBJECT ORIENTED METHODOLOGY .................................................................... 8

2.1 R&D PROJECTS HISTORY ......................................................................................... 13

2.2 TOWARDS AN INTEGRATED MODELING ENVIRONMENT ....................................... 16

2.3 THESIS ORGANIZATION ......................................................................................... 22

1 Modeling Concepts and Fundamental Algorithms ................................................. 25

1.1 DATA MODELING ........................................................................................................ 28

1.1.1 Cell class .................................................................................................................. 29

1.1.2 Zone class ................................................................................................................ 36

1.1.3 Coherence and Tolerance Concept ........................................................................ 41

1.1.4 Zone classification and Surface class ..................................................................... 42

1.2 COMPUTATIONAL TOPOLOGY ................................................................................... 44

1.2.1 Topological space ................................................................................................... 44

1.2.2 Structured and Unstructured Topology .................................................................. 46

1.2.3 Cell Connectivity .................................................................................................... 49

1.2.4 Node Topology ......................................................................................................... 54

1.2.5 Domains Connectivity ............................................................................................. 57

1.2.6 Cell Intersection Pattern ....................................................................................... 61

1.3 CONTINUITY ASSUMPTION .................................................................................... 72

1.3.1 Interpolation Method .............................................................................................. 73

1.3.2 Derived Quantity .................................................................................................... 85

1.4 EXTRACTION ALGORITHMS ..................................................................................... 90

1.4.1 Marching Cell Algorithm ....................................................................................... 90

1.4.2 Threshold Regions .................................................................................................. 96

1.4.3 Curve Extraction: section and isoline algorithm .................................................... 98

1.4.4 Point Extraction: local value algorithm .................................................................. 100

1.4.5 Particle Trace Algorithm ....................................................................................... 113

2 Adaptation of Visualization Tools ......................................................................... 121

2.1 INPUT DATA MODEL .................................................................................................. 124

2.2 SURFACE MODEL ..................................................................................................... 129

2.3 GEOMETRY REPRESENTATIONS ............................................................................. 132

2.4 QUANTITY REPRESENTATIONS ............................................................................. 134

2.4.1 Isolines ..................................................................................................................... 135

2.4.2 Quantity Fields and Thresholds ............................................................................. 135

2.4.3 Point based numerical probes ................................................................................ 138

2.4.4 Curve based numerical probes .............................................................................. 139

2.4.5 Surface based numerical probes ............................................................................. 143

2.4.6 Vector line numerical probes ................................................................................ 146

2.5 USER-CENTERED APPROACH ............................................................................... 148

2.6 INTERACTION MODELING ....................................................................................... 150

2.7 VIEWING SPACE AND NAVIGATION ...................................................................... 155

2.8 VISUALIZATION SCENARIOS ................................................................................... 159
3 Object-Oriented Software Development ......................................................... 167

3.1 SOFTWARE ENGINEERING MODEL ............................................................... 167
3.2 OBJECT ORIENTED CONCEPTS .................................................................. 170
  3.2.1 Objects and Classes .................................................................................. 171
  3.2.2 Messages and Methods .......................................................................... 173
  3.2.3 Inheritance ............................................................................................. 174
  3.2.4 Polymorphism ......................................................................................... 176
3.3 OBJECT ORIENTED PROGRAMMING IN C++ ............................................ 177
3.4 EXAMPLE OF OBJECT-ORIENTED SOFTWARE DEVELOPMENT .................. 179
  3.4.1 Requirements, Analysis and Design Example .......................................... 180
  3.4.2 Data Abstraction and Encapsulation ...................................................... 184
  3.4.3 Inheritance and Polymorphism ............................................................... 187
  3.4.4 Templates and Exception Handling ......................................................... 189
  3.4.5 Dynamic Memory Management ............................................................ 189
3.5 MAPPING ENTITY-RELATIONSHIP MODEL TO OBJECT-ORIENTED MODEL .... 191
3.6 MODEL-VIEW-CONTROLLER DESIGN ........................................................ 192
3.7 VISUALIZATION SYSTEM ARCHITECTURE ............................................... 196
  3.7.1 Model Layer – CFD, Continuum and Geometry classes ......................... 198
  3.7.2 View Layer – 3D Graphics Category ..................................................... 199
  3.7.3 Controller Layer – 3D and GUI input processing ..................................... 202
3.8 SOFTWARE DEVELOPMENT ENVIRONMENT .............................................. 205
  3.8.1 Software Reusability .............................................................................. 205
  3.8.2 Software Portability .............................................................................. 205
  3.8.3 Integrated Development Environment .................................................. 205

Conclusion and Future Developments ................................................................. 208

PASHA – PARALLEL CFVIEW – IMPROVING THE PERFORMANCE OF VISUALIZATION ALGORITHMS .......... 210
ALICE – QFVIEW – TOWARDS THE TRANSPARENT VISUALIZATION OF NUMERICAL AND EXPERIMENTAL DATA SETS .......................................................... 212
QNET-CFD – ON QUALITY AND THRUST IN INDUSTRIAL APPLICATIONS .................. 214
LASCOT – VISUALIZATION AS DECISION-MAKING AID ................................... 215
Serket – Security situation awareness ............................................................... 218
FUTURE DEVELOPMENTS ............................................................................... 219
Web3D for Collaborative Analysis and Visualization ........................................ 219
European Scientific Computing Organization .................................................. 220
Development trends in Interactive Visualization systems ................................... 220

References .......................................................................................................... 225

Appendixes ......................................................................................................... 234

LOOKUP TABLE AND ITS C++ IMPLEMENTATION FOR THE PENTAHEDRON CELL .............................. 234
RESEARCH PROJECTS TIMELINE ........................................................................ 239
AUTHOR’S PUBLICATIONS AT VUB ................................................................. 240
RESEARCH DEVELOPMENTS TIMELINE .......................................................... 242
SUMMARIZING THE CURRENT STATE-OF-THE-ART ............................................. 244
PERFORMANCE ANALYSIS ............................................................................. 246
  Theoretical Test Cases ..................................................................................... 246
  Hardware Environment ................................................................................... 247
  Algorithms Parameters ................................................................................... 248
  Measurements Timing ..................................................................................... 250
  Measurements Results .................................................................................... 250
Abstract

This thesis describes a novel approach to create, develop and utilize software tools for visualization in scientific and engineering applications. These Scientific Visualization (SV) tools are highly interactive visual aids which allow analysis and inspection of complex numerical data generated from high-bandwidth data sources such as simulation software, experimental rigs, satellites, scanners, etc... The data of interest typically represent physical variables -- 2- and 3-dimensional scalar, vector and tensor fields on structured and unstructured meshes in multidomain (multiblock) configurations. The advanced SV tools designed during the course of this work permit data extraction, visualization, interpretation and analysis at a degree of interaction and effectiveness that was not available with previous visualization techniques.

The Object-Oriented Methodology (OOM), which is the software technology at the basis of the approach advocated in this thesis, is very well adapted for large-scale software development: OOM makes SV tools possible and makes them a usable, innovative investigation instrument for the engineer and the researcher in all areas of pure and applied research. The advanced SV tools that we have developed allow the investigator to examine qualitatively and quantitatively the details of a phenomenon of interest, in a unified and transparent way. Our SV tools integrate several well-known algorithms -- such as the cutting plane, iso-surface and particle trace algorithms -- and enhance them with an ergonomic graphical user interface. The resulting SV system implements the reusability and encapsulation principles in its software components, which support both space discretization (unstructured and structured meshes) and continuum (scalar and vector fields) unconstrained by the grid topology. New implementation mechanisms applied to the class hierarchies have been developed beyond existing object-oriented programming methods to cover a broader range of interactive techniques. A solution was found to the problem of developing, selecting and combining classes as reusable components. The object-oriented software development life-cycle was mastered in the development of these classes, which were finally packaged in a set of original class libraries.

A main outcome of our approach was to deliver one of the first frameworks, which integrates 3D graphics and windowing behavior based on the software components implemented in C++ only. This framework ensures maximal portability to different hardware platforms and establishes the basis for reusable software in industrial applications, such as an integrated Computational Fluid Dynamics (CFD) environment (pre-post processing and the solver). The important outcome of this work is an integrated set of VS tools -- the Computational Field Visualization System (CFView) -- available for investigators in field physics in general, and specifically for CFD researchers and engineers. Several CFD examples are presented and discussed to illustrate the new development techniques for scientific visualization tools.
Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Charles Hirsch, for taking the risk of introducing an innovative software-construction methodology to the CFD community and for giving me the opportunity to master it during all these years. His probing remarks, his continuous support and advice spurred many thoughts and were always an incentive for me to pursue excellence in this work. Without Prof. Chris LACOR, this thesis would not have come to completion, and I deeply thank him for his support and encouragement in the finalization phase of this thesis.

Thanks go to all my colleagues, former or present members of the Department of Mechanical Engineering, the former Department of Fluid Mechanics of the Vrije Universiteit Brussel, who contributed to the development of CFView: Michel Pottiez, Vincent Sotiaux, Cem Dener, Marc Tombroff, Jo Decuyper, Didier Keymeulen, Jan Torreele, Chris Verret, as well as to the developers at NUMECA: Jorge Leal Portela, Etienne Robin, Guy Stroobant and Alpesh Patel, who pursued the development of CFView and made it an advanced scientific visualization product for turbo-machinery applications.

Special thanks go to the many researchers and engineers who used CFView: Shun Kang, Peter Segaert, Andreas Sturmayr, Marco Mulas, Prasad Alavilli, Zhongwen Zhu, Peter Van Ransbeeck, Erbing Shang, Nouredine Hakimi, Eric Lorrain, Benoît Léonard, François Schmitt, Evgeni Smirnov, Andrei Khodak, Martin Aube, Eric Grandry, for their valuable comments and suggestions. Since 1988, when I joined the Department, I have had many useful and challenging interactions with Francisco Alcrudo, Antoine van Marcke de Lummen, Guido Van Dyck, Peter Grognard, Luc Dewilde, Rudy Derdelinckx, Steven Haesaerts, Famakan Kamissoko, Hugo Van Bockryck, Karl Pottie, Wim Teugels, Wim Collaer, Kris Sollie and Pascal Herbosch; these exchanges of views contributed to making this work richer and better. I happily acknowledge the inputs of the younger group of scientists at our Department and their ideas on novel visualization aspects in acoustics, combustion, medicine and environment domains: Stephan Geerts, Jan Ramboer, Tim Broeckhoven, Ghader Ghorbaniasl, Santhosh Jayaraju, Mark Brouns, Mahdi Zakyani and Patrik Widera. I am indebted to Jenny D’haes, Alain Wery and Michel Despees, without whom my daily work at the Department would not be possible.

I am pleased to mention the contribution of Prof. Jacques de Ruyck on PLOT83, elements of which were applied in CFView. I also thank Prof. Herman Deconinck for offering me the opportunity to give a lecture on object-oriented programming in computer graphics at von Karman Institute in 1991, which provided momentum to my work to this exciting research area. For our interactions on unstructured modeling, I thank Jean-Marie Marchal and gratefully remember the late Yves Rubin, whose suggestions prompted the development of CFView in the finite-element-method domain. Special thanks go to Patrick Vankeirsbilck for many enlightening discussions on object-oriented programming practice.

I would like to thank Koen Grijspeerdt for our teamwork in the IWT-funded LCLMS project, and for his effort in introducing CFD and visualization in food-production applications.

Special thanks go to Birinchi Kumar Hazarika for the work in the EC-funded ALICE project and to Cristian Dinescu, John Favaro, Bernhard Sünder, Ian Jenkinson, Giordano Tanzini, Renato Campo, Gilles Gruez, Pasquale Schiavo and Petar Brajak.

I take the opportunity to thank my collaborators Danny Deen, Emil Oanta and Zvonimir Batarilo for successful development of the Java visualization components in the SERKET project. I would like to thank Prof. Jan Cornelis and his collaborators Hichem Sahli, Rudi Deklerk and Peter Schelkens for their contributions when extending scientific visualization to general-purpose information systems. Special thanks also go to Claude Mayer, François-Xavier Josset, Tomasz Luniewski, Jef Vanbockryck, Claude Desmet, Karel Buijsse, Christophe
Herreman, Luc Desimpelaere and Richard Aked for their help in shaping the visualization development in ITEA projects.

Two EU-funded TEMPUS projects, made it possible for me to interact again with Croatia and Macedonia, and my thanks go to Prof. Zdravko Terze and Prof. Milan Kosevski for encouraging me to complete this thesis.

Special thanks go to Bernard Delcourt for his thorough proof-reading and improving the written English, and sharing with me the last moments before the publishing of this thesis.

The funding of the European Commission (EC) and the Flemish institute for Innovation and Technology (IWT) is gratefully acknowledged; the LCLMS, ALICE, LASCOT, QNET-CFD and SERKET projects have been instrumental in allowing me to carry out my research. I am grateful to Vrije Universiteit Brussel for providing the necessary research and computer facilities, not only to accomplish this work, but also to complete the engaged projects.

I would like thank my parents for their moral and financial support, which made it possible for me to come to Belgium; I will always remember my mother for her unfailing enthusiasm for research work, an attitude she has passed on to me and for which I will ever be grateful. I thank my father and brother who kept chasing me up and pushing me to complete this work.

Finally and most importantly, I wish to thank my wife and children for their love, patience, support and encouragement during these many years, months, weekends, days and evening hours that went into this longer-than-expected undertaking.

Dean Vučinić
Brussels, July 2007
Nomenclature

∀ for all
∃ there exist
= equals
∑ summation
Π product

Expressions:

\( s_{ij} \) metric tensor components \((i,j = 1,2,3)\)
i, j, k grid indices in u, v and w directions
I, J, K grid numbers in i, j and k coordinate index directions
\( J \) Jacobian of transformation
\( G \) metric tensor
n normal distance
\( n \) normal vector
p grid point coordinates
r, x position vector
S structured grid
A transformation matrix
u, v, w parametric curvilinear coordinates
U unstructured grid
x, y, z Cartesian coordinates
\( X \) hybrid grid

Symbols:

\( \delta \) Kronecker delta; search radius
\( \partial \) partial differential operator; boundary
\( \epsilon \) error of numerical solution
\( \nabla \) Gradient operator
\( \nabla \cdot \) Divergence operator
\( \nabla \times \) curl of vector
\( \nabla^2 \) Laplacian operator
\( \Delta \) forward difference operator
\( \Delta u, \Delta v, \Delta w \) spacing in parametric coordinates
\( \Delta x, \Delta y, \Delta z \) spacing in Cartesian coordinates

Subscripts and superscripts:

i, j, k position indices for structured meshes
u, v, w direction of differentiation or interpolation in parametric space
1, 2, 3 space coordinate axis

Abbreviations:

ADT Abstract Data Type
AFX Animation Framework Extension
AI Artificial Intelligence
ANC Automatic Naming Convention
ATDC After Top Dead Centre
AVS Advanced Visual Systems
BC Boundary Condition
B-rep Boundary Representation
CA  Crank Angle
CAD  Computer Aided Design
CAE  Computer Aided Engineering
CFD  Computational Fluid Dynamics
CG  Computer Graphics
CFView  Computational Flow Field Visualization
CON  Connection BC
CPU  Central Processing Unit
DFD  Data Flow Diagram
DNS  Direct Numerical Simulations
DPIV  Digital Particle Image Velocimetry
EFD  Experimental Fluid Dynamics
ERM  Entity Relationships Model
EXT  External BC
FD  Finite Difference
FE  Finite Element
FEA  Finite Element Analysis
FO  Function Oriented
FV  Finite Volume
GPU  Graphics Processing Unit
GUI  Graphical User Interface
HOOPS  Hierarchical Object Oriented Picture System
HWA  Hot wire Anemometry
IME  Integrated Modeling Environment
INL  Inlet BC
J2EE  Java 2 Platform, Enterprise Edition
JOnAS  Java Open Source J2EE Application Server
KEE  Knowledge Engineering Environment
LAN  Local Area Network
LG  Local to Global index mapping
LDV  Laser Doppler Velocimetry
LSE  Large Eddies Simulation
LSV  Light sheet visualization
MB  Mega Bytes
MFLOPS  Millions of Floating Point Operations per Second ("MegaFlops")
MIPS  Millions of Instructions per Second
MIMD  Multiple instruction, multiple data
MPEG  Moving Picture Expert Group
MVC  Model View Controller
MVE  Modular Visualization Environments
OO  Object Oriented
OOM  Object Oriented Methodology
OOP  Object Oriented Programming
OOPL  Object-Oriented Programming Language
OUT  Outlet boundary condition
PC  Personal Computer
PDE  Partial Differential Equations
PER  Periodic BC
PIV  Particle Image Velocimetry
PHIGS  Programmer’s Hierarchical Interactive Graphics System
PVM  Parallel Virtual Machine
QFView  Quantitative Flow Field Visualization
List of Figures

Figure 1: The scientific visualization role _______________________________________________________ 2
Figure 2: The Scientific Visualization Model ____________________________________________________ 4
Figure 3: The Visualization Data Sets __________________________________________________________ 5
Figure 4: Integrated Computational Environment ________________________________________________ 6
Figure 5: CFView the scientific visualization system _____________________________________________ 9
Figure 6: The OpenDX Application Builder ______________________________________________________ 10
Figure 7: VisAD application example ____________________________________________________________ 11
Figure 8: The integrated modeling environment from Dassault Systèmes and ANSYS, Inc ________________ 12
Figure 9: The comparison of Hardware/Software productivity ________________________________________ 13
Figure 10: Graphics Engine as combine software/hardware solution _________________________________ 15
Figure 11: Software Components Distribution ____________________________________________________ 15
Figure 12: QFView Web Interface __________________________________________________________________ 17
Figure 13: Use of EFD and CFD tools _____________________________________________________________ 18
Figure 14: Laminar flow at Reₜ = 60: DPIV of nearly axisymmetric flow, LSV of vortex shedding and CFD at various degrees of non-axisymmetric flow ___________________________________________ 18
Figure 15: PIV system at VUB ___________________________________________________________________ 19
Figure 16: Flow pattern for 90°ATDC: (a) visualization at 20 rev/min, valve lift = 10 mm (b) Average velocity field at 5 rev/min, valve lift = 10 mm _____________________________________________________________ 19
Figure 17: Turbulent kinetic energy field at 90°ATDC at 5 rev/min, valve lift = 10 mm ______________________ 20
Figure 18: Average vorticity field at 90°ATDC at 5 rev/min, valve lift = 10 mm __________________________ 20
Figure 19: Experimental and CFD model for flow analysis between corrugated plates ____________________ 21
Figure 20: Workflow for the integration of EFD and CFD simulations ____________________________________ 21
Figure 21: Example of an Integrated Modeling Environment [57] ______________________________________ 22
Figure 22: Example of the 3D virtual car model testing ________________________________________________ 23
Figure 23: Software model as communication media in the software development process __________________ 25
Figure 24: Entity-Relationship Model _______________________________________________________________ 26
Figure 25: Data model decomposition ___________________________________________________________________ 28
Figure 26: Cell classification ________________________________________________________________________ 30
Figure 27: Developed cell ERM _____________________________________________________________________ 31
Figure 28: 1D & 2D Cell topologies ____________________________________________________________________ 32
Figure 150: Vector Line menus and representations
Figure 117: Geometry/Surface menu
Figure 109: Main Quantity menu
Figure 90: Cell ray intersection
Figure 149: Combined use of Particle trace, Cutting plane and Isosurface probes
Figure 148: Several isosurface representations of the Temperature field around the airplane
Figure 146: Cutting planes representations in 3D
Figure 145: Cutting Plane dialog-box
Figure 144: Vector Local Profiles
Figure 143: Local profile representation for boundary layer analysis
Figure 142: Isolines and Surface Particle Paths in 3D
Figure 141: Scalar and Vector curve based extractions in 2D
Figure 140: Cartesian plot of the shock wave location in 3D
Figure 139: Scalar distribution along solid boundaries and sections
Figure 138: Cartesian Plot menu
Figure 137: Render menu
Figure 136: Local isolines, scalars and vectors assisted with coordinate axis tool
Figure 135: Vector Thresholds on cutting planes
Figure 134: Structured and unstructured vector fields
Figure 133: Vector Field dialog-box
Figure 132: Threshold color contours
Figure 131: Color contours based on different rendering algorithms
Figure 130: Color Contours menu
Figure 129: Isolines menu item and dialog-box
Figure 128: Isolines representations for 2D and 3D geometries
Figure 127: Scalar and vector representation menus
Figure 126: Rendering of the space shuttle
Figure 125: Plot Data menu
Figure 124: Geometry repetitions types: mirror, translation and rotation
Figure 123: Geometry menu
Figure 122: Geometry menu
Figure 121: Surface dialog-box showing the cutting plane and isosurface instances
Figure 120: Cutting plane and isosurface examples for structured and unstructured meshes
Figure 119: Create Surface dialog-box
Figure 118: Structured multiblock surface patch manipulation
Figure 117: Domains connectivity for structured 3D multiblock grids
Figure 116: Initial representations: boundaries in 2D and solid surface boundaries in 3D
Figure 115: Domains connectivity for structured 3D multiblock grids
Figure 114: Unstructured topology for 2D and 3D geometry
Figure 113: Structured topology for 2D and 3D geometry
Figure 112: Plot Data menu
Figure 111: Validation Data menus
Figure 110: Field and Solid Quantity menu
Figure 109: Main Quantity menu
Figure 108: Local values, Isolines and Particle paths representations in 3D
Figure 107: Local profile and Particle paths for a 2D airfoil
Figure 106: The map of a cell boundary point between connected cells in 2D
Figure 105: The map of a cell boundary point between connected cells in 3D
Figure 104: The cell boundaries parametric space in 3D
Figure 103: The cell boundaries parametric space in 2D
Figure 102: Tangency condition for the vector line analytical and numerical treatment
Figure 101: Possible positions of the point leaving the cell in the neighborhood of the pyramid
Figure 100: Possible positions of the point leaving the cell in the neighborhood of the pyramid
Figure 99: Possible positions of the point leaving the cell in the neighborhood of the prism
Figure 98: Possible positions of the point leaving the cell in the neighborhood of the tetrahedron cell
Figure 97: Possible positions of the point leaving the cell in the neighborhood of the quadrilateral cell
Figure 96: Possible positions of the point leaving the cell in the neighborhood of the triangle cell
Figure 95: Point location on cell boundaries
Figure 94: Boundary normals for different parametric dimensions
Figure 93: Rays and intersections with extended cell boundary
Figure 92: Plane line intersection.
Figure 91: Intersection parameter description
Figure 90: Cell ray intersection
List of Tables

Table 1: Layered Software Architecture ____________________________________________ 14
Table 2: SEGMENT skeleton table ________________________________________________ 32
Table 3: TRIANGLE skeleton table ________________________________________________ 32
Table 4: QUADRILATERAL skeleton table __________________________________________ 32
Table 5: TETRAHEDRON skeleton table ____________________________________________ 34
Table 6: PYRAMID skeleton table __________________________________________________ 34
Table 7: PENTAHEDRON skeleton table ____________________________________________ 34
Table 8: HEXAHEDRON skeleton table _____________________________________________ 35
Table 9: Structured zone parameterization _________________________________________ 47
Table 10: C++ implementation of the hashing value ________________________________ 53
Table 11: Boundary indexing for 2D and 3D structured grids ____________________________ 57
Table 12: Domain connectivity specification in 2D ________________________________ 58
Table 13: Domain connectivity specification in 3D ________________________________ 59
Table 14: The WEB model record ________________________________________________ 61
Table 15: WEB model of the tetrahedron __________________________________________ 63
Table 16: The lookup table for the tetrahedron ____________________________________ 68
Table 17: Polygon Subdivision __________________________________________________ 68
Table 18: Records from hexahedron lookup table with polygon partitions and multiple connected regions _________________ 69
Table 19: Lookup table for the triangle __________________________________________ 70
Table 20: Lookup table for the quadrilateral _______________________________________ 71
Table 21: Shape function for 3D isoparametric mapping ____________________________ 78
Table 22: Reduction of multiplication operations ________________________________ 80
Table 23: Triangle truth table __________________________________________________ 104
Table 24: Triangle constrains path ________________________________________________ 106
Table 25: Quadrilateral truth table ______________________________________________ 106
Table 26: Quadrilateral constrains path ___________________________________________ 107
Table 27: The mapping procedure of a cell boundary point between connected cells in 3D ________________________________ 119
Table 29: Standard notation for boundary conditions ________________________________ 128
Table 30: Comparison of classical and user- centered approach _______________________ 148
Table 31: Software quality factors ________________________________________________ 169
Table 32: Graphics primitives for different geometries and text ______________________ 201
Table 33: Average times (s) for Sequential, SIMD and MIMD implementations of Cutting Plane and Isosurface algorithms (wall-clock time) ____________________________________________________________ 211
Table 34: The lookup table for the pentahedron ____________________________________ 234
Table 35: Research Projects Timeline ____________________________________________ 239
Table 36: Average times for Cutting Plane (wall-clock time in seconds) ____________ 251
Table 37: Average times for Isosurface (wall-clock time in seconds) ________________ 251
Table 38: Average times for Particle Trace (wall-clock time in seconds) ____________ 251
Table 39: Evolution of the execution times in seconds with the number of particles used ____________________________________________________________ 252
Table 40: Execution times in seconds for Isosurface on MIMD for different machine configurations (wall-clock time) with varying number of processors ____________________________________________________________ 252
**Introduction**

Fluid motion is studied in Fluid Dynamics [1, 2] by performing experimental and computational simulations that researchers analyze in order to understand and predict fluid flow behaviors. This scientific process yields large data sets, resulting from measurements or numerical computations. Scientific visualization comes naturally in this process as the methodology that enhances comprehension and deepens insight in such large data sets. The term *Scientific Visualization* was officially introduced and defined as a scientific discipline in 1987 at SIGGRAPH[3] and contributes to the role of computing, as quoted by Richard Hamming:

*The purpose of computing is insight, not numbers.*

Scientific visualization [4, 5] is usually performed through specialized software, which combines visualization techniques to display and analyze scientific data. The scientific visualization methodology defines methods to manipulate and convert data into comprehensible images. The scientific visualization process starts with the transformation of data sets into geometric abstractions, which are further processed in displayable images, created by computer graphics algorithms [6, 7]. Finally the human vision, possessing the highest-bandwidth of human’s information input, is exploited to understand the computer generated images.

Computational Fluid Dynamics (CFD) simulations are an important source of scientific and engineering data, for example, computations of complex three-dimensional (3D) internal flows in Turbomachinery and external flows around complete aircraft or space shuttle configurations. The numerical simulation software used in the industry, typically based on 3D Navier-Stokes solvers, requires easy and efficient visualization and presentation tools to process the flow fields. The numerical simulation environment is expected to be ergonomic, so as to incite the CFD specialist to investigate complex flow topologies and explore the physics of fluid flows in a user-control manner. A key objective of the visualization process is to enable the creation of meaningful images from large amounts of data.

In order to develop a scientific Visualization Software (VS) it is necessary to combine Computer Graphics (CG) and User Interface (UI) design know-how with engineering knowledge. Thus, we need to consider and integrate the mentioned methodical domains, when addressing the software development of scientific visualization software. The well adapted approach to address scientific visualization issues is through the introduction of interactive visualization tools for the analysis and interpretations of numerically generated data, which need to be developed by taking into account and combining computer hardware (processor, disks, graphics boards) and computer software (graphics application programming interface (API) and scientific visualization methods).

Scientific visualization methods give the possibility to the researcher to analyze the same problem in a variety of ways. They promote the concept of *virtual laboratory* where the user performs *virtual measurements*. These virtual measurements are the product of the *interactive visualization* process that the user applies on the data he/she analyzes. For example, the interaction may lead to the researcher locating a flow re-circulation area, a region of high turbulence, or the points where the value of enthalpy reaches its minimum and maximum values. Usually, the goal is not only to see the flow pattern, but also to understand why and how it develops. The possibility to apply several visualization tools in an interactive way makes it possible to maybe uncover the problem under investigation. The exploration of numerically generated data leads to the concept of *virtual instrument*, namely a *software probe* that can be interactively manipulated. Two important questions are to be answered:

*How does software probe (virtual instrument) measure a value?*

- Filtering and searching through numerical data.

*How does it graphically display a measured value?*

- Applying computer graphics techniques.
As shown in Figure 1, numerically generated data are the main input to the visualization system. The data sources are experimental tests and computational models which yield high-resolution, multi-dimensional data sets. Such large and complex data sets may consist of several scalar, vector and/or tensor quantities defined on 2D or 3D geometries; they become even larger when time-dependency or other specialized parameters are added to the solution. Fast and selective extraction of qualitative and quantitative information is important for interpreting the visualized phenomena. The performance (response time) of the visualization system becomes critical when applying iterative optimization procedure to the model under analysis. An adequate computer system performance (response loop) must be in place, in order to match the activity of the human visual system and the computer display of the extracted flow features to enable the user to enjoy the benefits of a truly interactive visualization experience. The visualization system must also provide extensive interactive tools for manipulating and comparing the computational and experimental models. For example, to lower the costs of a new product development, we can reduce the range of experimental testing and increase the number of numerical simulations, provided we are able to effectively exploit the existing experimental database.

The main role of SV is to present the data in a meaningful and easily understandable digital format. Visualization can be defined as a set of transformations that convert raw data into a displayable image; the goal is to convert the raw information into a format understandable by the human visual system, while maintaining the completeness of the presented information expected by the end user. In our work, we used Vector Graphics (VG) for applying colors and plotting functionality on geometrical primitives such as points, lines, curves, and polygons. This is in contrast to the Raster Graphics (RG) approach, which represents images as a collection of pixels. Other scientific visualization techniques rely on comparison and verification methods based on RG high-resolution images. Such visualization systems to analyze RG images, for example coming from satellites and scanners are not developed in this work, but their results are integrated, as part of the performed physical experiments.

The present work contributed to the development of Scientific Visualization by addressing the following questions:
**Why scientific visualization?**

- to help scientists and engineers visualize and explore their results in the simplest possible way,
- to improve scientific visualization techniques,
- to provide new and better ways of displaying data,
- to discover new visual interpretations through image synthesis,
- to standardize knowledge of scientific visualization,
- to promote and improve software development platform for industrial applications,
- to discover new phenomena in the regions where measurements cannot be taken,
- to validate computational and experimental results,
- to improve the learning process, as a pedagogical tool for aiding students, engineers in the fluid flows modeling.

**How to do scientific visualization?**

- Utilizing interactive scientific visualization software.

**How to develop interactive scientific visualization software?**

This is the fundamental question which was addressed in the present work:

- multidisciplinary research, CFD and Computer Science
- scientific visualization tools
- object oriented methodology and software engineering,
- new ways of interacting with numerical codes and computer environment,
- new algorithms improving user perception (color, shape),
- graphical user interface design
- application of windowing (X11, Windows) and graphics standards (PHIGS, PEX, OpenGL)
- implementations based on object oriented programming languages in C++ and Java
- interactive tools for large data sets applying network computing with parallel machines

A blend of many developments in computational and computer graphics hardware and software has made it possible to design interactive SV software. Today computer hardware and software give enough performance and throughput for graphical and computational tasks to achieve the expected level of functionality, namely:

**Computational task:** search and extraction algorithms that can identify, compute and extract the geometrical and quantitative data from selected data sets.

**Visualization task:** 3D computer graphics display and user-interaction with the graphical objects.
**Scientific Visualization Model**

The scientific visualization model defines a series of transformations which convert data into images displayed on the computer screen, as shown in Figure 2. The model encompasses four visualization data types: simulation, derived, graphics and displayable data. The visualization process is defined as a set of mappings, which transform the visualization data types from one data type to another.

<table>
<thead>
<tr>
<th>DATA</th>
<th>TRANSFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulated</td>
<td>extraction refinement</td>
</tr>
<tr>
<td>Derived</td>
<td>enhancement enrichment</td>
</tr>
<tr>
<td>Graphical</td>
<td>rendering</td>
</tr>
<tr>
<td>Displayable</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2: The Scientific Visualization Model*

The first *extraction/refinement* transformation is applied to reduce the input simulation data to a sizeable and formatted data set, in order to make it acceptable for subsequent processing steps. Typically, the simulated data are the values of some physical variable at discrete locations of the solution domain. It is often necessary to use some form of interpolation of the given values to obtain the new data at other specified points. For example, it may be necessary to calculate the values of the variable on an arbitrary curve which is itself defined by a set of discrete points. Such process requires the creation of additional data sets in order to approximate parts of the continuum domain. For example, an additional data set would be needed to define a set of surface normals, which must be available to the rendering transformation for it to display the surface as a smooth surface, i.e. without visible edge discontinuities.

The second step is an *enhancement/enrichment* transformation, which relates data values to graphical attributes such as color, transparency, luminosity, texture and reflectance. Usually, the mapping functions between values and graphical attributes are interactively manipulated by the user. Nonlinear mappings tend to be more useful than linear ones, as they can better reveal details of the visualized phenomena. As there are no methods capable of identifying the ‘best’ transfer functions, the user-driven interactive approach can be quite effective in exploring different transfer functions.

The third and last step consists of *rendering* transformations which produce displayable images. Typical rendering operations include view mappings such as rotation, translation and scaling, followed by perspective mapping and clipping. The rendering of 3D volumetric effects include hidden line/surface removal, shading, anti-aliasing and so forth. More elaborate algorithms deal with transparency and shading (ray tracing).
Figure 3: The Visualization Data Sets
The SV model needs to be well understood by the user -- the investigator -- in order for him/her to correctly interpret the displayed data. The user must be able to fully control the amount of visualized information in order to extract meaningful information from the data without being overloaded with graphical content.

Figure 3 shows examples of results of the visualization process. The solid model, mesh grid and computed physical quantities are initial simulation data. Applying the next extracting transformation permits the interactive generation of user-defined surface or curves. The mirror surface extracted from the double ellipsoid test case represents the reduced data set. In the third step, the image is enriched and enhanced by a color mapping. Computer-graphics primitives, such as polygonal meshes, store the data in the format treatable by the underlying computer graphics engine. The final rendering transformation treats the 3D graphical objects with viewing and lighting transformations.

**Scientific Visualization Environment**

The scientific visualization system must be an integrated part of the computational environment (Figure 4) if it is to efficiently assist the scientist/engineer in his/her numerical simulations data analysis. Such software systems are used today in research laboratories and in the industry. In the industry, visualization is used to gain a more quantitative understanding of the simulated phenomena (ex. aerospace product design); the results of visualization are also used in management and commercial presentations. In contrast, in the research laboratory, scientists develop codes and try to understand qualitatively how the simulation algorithms behave; in this context, they tend to use SV as a debugging tool. In both cases, the computational environment includes software that supports geometrical definition (as in CAD systems), mesh generation (pre-processing), supervision of the simulation (co-processing) and display and/or analysis of results (post-processing).

It is understood that in order to be effective, CFD software must be integrated into the (usually large) software platforms used by the research institutes or industries. Clearly, the success of such software integration depends on the sound application of software engineering methods, not on advances in CFD models or algorithms. Building such systems requires the coordinated, multi-disciplinary effort of CFD and computer experts. Developing integrated computation/visualization platforms has an impact on the engineering of simulation software, which must interoperate with the visualization software; this, in turn, places new constraints on the computer hardware which must meet the computational and graphical demands of the integrated software solution.

![Integrated Computational Environment](image)
The CFD simulation process consists of:

- geometry definition
- grid generation
- numerical simulation
- solution analysis and comparison

In each of these phases, applying scientific visualization is of importance.

Geometry definition involves the creation of a spatial model of the object of interest (for example, a car or a part of a plane) which includes defining the shape of the object. Visualization is applied here to explore (variants of) the resulting model and to validate the input data received from a CAD like system.

Grid generation involves the discretization of the problem domain (on a structured or unstructured grid). Visualization and graphical input techniques are applied to construct geometrical models, to evaluate the results of the modeling, to detect errors and to control and monitor the grid generation process.

Numerical simulation generally involves discrete representations of the field variables and approximate statements of the boundary/initial value problem. The analytical mathematical problem is transformed into an approximate mathematical problem that can be solved using numerical methods. In nonlinear and transient solutions, visualization is used to monitor the behavior and evolution of the simulation. Here, visualization can rapidly reveal unstable and divergent (erroneous) solutions. Visualization can also help the investigator to tune the solution parameters interactively so as to maintain the accuracy and stability of the computation. Clearly, visualization modifies the very nature of the investigation process by making it highly interactive, providing more control to the engineer/scientist who can better tune his/her computational experiment, optimize the use of computer resources and dramatically reduce the time to results.

The result of the numerical simulation is a solution, i.e. the field values of some ‘primary’ quantities. In the solution analysis phase, ‘secondary’ (or ‘derived’) quantities are computed using data extraction and post-processing calculations; the secondary data are essential for validating and presenting the results of the simulation.

Current best practice suggests four visualization phases, involving different performance and memory requirements for computation, graphics, mass storage and communication:

- pre-processing
- co-processing
- post-processing
- interactive steering

Pre-processing verifies input settings before the simulation is computed. The post-processing mode treats the simulation output data stored for visualization. This mode is often appropriate since the cost, time or effort of repeating the simulation is usually larger than the cost of storing the results. It supports interactive visualization and real-time animation (useful when the simulation itself cannot be performed at animation rates). During the co-processing, the visualization system monitors in real-time the evolution of the simulation in progress by displaying the computed data at each iteration. Data histories can be accumulated to permit animation of partial results. This allows the user to monitor the behavior of the solution, to identify convergence or other problems and to abort the simulation if errors occur. The fourth mode -- the most demanding one as far as computer-resources are concerned --, is called interactive steering[8]; it combines simulation and visualization in a system-user closed loop where the user can interactively modify the simulation parameters with immediate visual feedback. A ‘data accumulator’ can be added to the pipeline at the end of the visualization mapping to
store intermediate results. If the system’s rendering module is capable of real-time animation, the accumulated data can be visualized while the main numerical computation continues in parallel.

Interactive visualization accelerates the CFD design cycle by allowing the user to ‘jump’ at will between the various phases so as to optimize his/her CFD analysis. The user conducts the investigation in a highly interactive manner, can easily compare variants of a simulation/analysis and may intuitively develop a deep understanding of the simulation and of the calculation details. An example of an integrated environment application is the ‘Virtual Wind Tunnel’ [9], which reproduces a laboratory experiment in a virtual reality environment, where a virtual model can be created and put to test with dramatic cost and time savings compared to what is done in the ‘real’ laboratory.

**Scientific Visualization Software - state of the art**

SV software has progressed enormously during the past two decades. One reason is the exponential increase in the power of the computer (central and graphical processors), which has led to today’s low-cost PCs providing as much power as the high-end mainframes of some years ago. Development of advanced SV tools is no longer the prerogative of specialized labs with costly computer equipment. Yet, there is an un-diminished demand for new visualization-enabled software, driven by continuous hardware changes and the emergence of new software platforms. Interactive visualization remains a key element of advanced engineering/scientific software, and their design must account for this fact. There are presently many commercial interactive visualization products on the market which provide SV functionality with increasing success. Such visualization systems are widely used in application areas as diverse as nuclear energy exploration and atmospheric research. In the field of engineering, such products are commonly used to visualize flow patterns and stress fields, and generally to study large multi-dimensional data sets. SV applications are used in many industries including aerospace, medicine, power production, shipbuilding, geophysics, automotive, electronics, oil, agriculture, food production, etc. ... SV applications are now ubiquitous in engineering and science, be it in:

- Fluid Mechanics,
- Structural Analysis,
- Electromagnetic,
- Thermodynamics,
- Nuclear Physic, etc.

For the sake of completeness, let us mention that SV has been (and is) instrumental in advancing the state of the art in industrial applications involving fluid flow modeling, such as:

- Aerodynamics of trains, cars and airplanes.
- Hydrodynamics of ships and floating structures.
- Flow in turbo-machinery and torque converters.
- Cryogenic rockets, combustion chambers simulations.
- Flow in manifold, pipes and machinery.
- Medical researches, circulation of blood in veins.

It is evident that advances in CFD software are driven by demands from many application areas, which in turn places requirements on the associated visualization software. Today, visualization software solutions with interactive 3D graphics capabilities can be categorized into four groups:
1. Visualization Applications
2. Modular Visualization Environments
3. Visualization Toolkits
4. Integrated Modeling Environments

1. Stand alone visualization applications are software solutions which offer direct functionality to the user, who is responsible for defining the data set to be loaded for visualization. Known visualization applications in CFD and Finite Elements Analysis (FEA) engineering are as follows:

   - EnSight from CEI [10],
   - FieldView from Intelligent Light’s[11, 12]
   - TecPlot from Amtec Engineering Inc.[13]
   - CFView from NUMECA [14]
   - PLOT 3D, NASA [15]
   - FAST (Flow Analysis Software Toolkit),NASA [16]
   - VISUAL2-3 from MIT [17],
   - FLOVIS from CIRA
   - HighEnd from DLR, [18].
   - ParaView from VTK [19]
   - VisIt from Lawrence Livermore National Laboratory [20]
Such programs are appropriate for users who need off-the-shelf visualization functionality. Such software implements the ‘event-driven’ programming paradigm which is suitable where all functions are launched by the user interacting at with the *Graphical User Interface (GUI)*. This is the case for *CFView* [21], see Figure 5, a scientific visualization application developed by the author over the 1988-98 period. *CFView* started as an academic application in 1988 and was continuously upgraded in the following years. In the mid 90’s, *CFView* was taken over by the VUB spin-off company NUMECA and integrated in ‘FINE’, NUMECA’s engineering environment. FINE is an environment that nicely illustrates the variety of visualization tasks that need to be performed to solve an engineering problem, especially addressed to turbomachinery applications.

2. **Modular Visualization Environments (MVE)** are programs often known as ‘visualization programming environments’; examples are[22]:
   - Advanced Visual Systems AVS [23],
   - *Iris Data Explorer from Silicon Graphics*[22, 24],
   - *OpenDX the IBM’s Data Explorer*[25],
   - *PV Wave from Visual Numeric* [26].

Their most significant characteristic is the *visual programming* paradigm. Visual programming intends to give users an intuitive *GUI* for them to build customized visualization applications. The user graphically manipulates programming modules displayed as boxes, which encapsulate the available functionality. By inter-connecting boxes, the user defines the data stream from one module to another, thereby creating the application. The *MVE* can be viewed as a ‘visualization network’ with predefined building blocks, and which often needs to be quite elaborate in order to be useful to the user. The freedom given to the users to design their own visualization applications is the strength of so-called ‘application builders’. This class of software implements the ‘data flow paradigm’, with the drawback that iterative and conditional constructs are difficult to implement. For example, *PV Wave* uses an interactive fourth-generation programming language (*4GL*) for application development, which supports conditional logic, data sub-setting and advanced numerical functionality in an attempt to simplify the use of such constructs in a visual programming environment. The interactive approach is usually combined with a script-oriented interface, and such products are not easy to use ‘right out of the box’ and have a longer learning curve than stand-alone applications.

![Figure 6: The OpenDX Application Builder](image-url)
There is an ongoing debate on whether the ‘best’ way to procure visualization software is to use stand-alone applications or to build applications using MVEs. Time has shown that both approaches are equally accepted as there is no alternative. The approach that we chose to follow in our work is a compromise between the two options. The GUI of our CFView software looks very much like that of a stand-alone visualization application; internally though, CFView is an object-oriented system which has the flexible, modular architecture of an application builder. This is to say that a new component can be integrated in the core application structure with a minimum coding effort; also, that the propagation effects resulting from the modification are kept limited.

3. Visualization Toolkits are general-purpose object-oriented visualization libraries, usually present as background components of SV applications. They emerged in the mid 1990’s, and the two representative examples are VTK[27] and VisAD[28]:

- The Visualization ToolKit (VTK) is an open-source software system for 3D computer graphics, image processing and visualization, now used by thousands of researchers and developers in the world. VTK consists of a C++ class library and several interpreted interface layers including Tcl/Tk, Java, and Python. VTK supports a wide variety of visualization algorithms (including scalar, vector, tensor, texture and volumetric methods), advanced modeling techniques (such as implicit modeling, polygon reduction, and mesh smoothing, cutting, contouring and Delaunay triangulation). In addition, dozens of imaging algorithms have been directly integrated to allow the user to mix 2D imaging / 3D graphics algorithms and data.

- The VISualization for Algorithm Development (VisAD) is a Java component library for interactive and collaborative visualization and analysis of numerical data. VisAD is implemented in Java and supports distributed computing at the lowest system levels using Java RMI distributed objects. VisAD’s general mathematical data model can be adapted to virtually any numerical data that supports data sharing among different users, different data sources and different scientific disciplines, and that provides transparent access to data independent of storage format and location (i.e., memory, disk or remote). The general display model supports interactive 3D (Figure 7), data fusion, multiple data views, direct manipulation, collaboration, and virtual reality. The display model has been adapted to Java3D and Java2D, and virtual reality displays.

![Figure 7: VisAD application example](image-url)
Integrated Modeling Environments (IME) is software that combines two or more engineering applications and visualization systems to solve a multi-disciplinary problem. For example, the naval architect shapes the ship hull in order to reduce the ship’s hydrodynamic drag, while the stress engineer calculates the ship’s steel structure. Both use visualization to analyze the data generated by the hydrodynamics and stress calculation solvers. The visualization software may be able to process the CFD flow-field solver data and the FEA stress-field solver data in a unified manner, giving the two engineers the possibility to work on compatible, interfacing 3D representations of the hydrodynamic and structural problems. An example of such integration is the Product Life-cycle Modeling (PLM) developed by Dassault Systèmes and the CFD solver technology developed by ANSYS, Inc, where the FLUENT CFD flow modeling approach is integrated in CATIA CAD tools throughout the whole product lifecycle [29].

*Figure 8: The integrated modeling environment from Dassault Systèmes and ANSYS, Inc*
**Object Oriented Methodology**

Computer hardware has improved drastically in quality and performance in the last 30 years, much faster than software quality and complexity. The trend is drawn qualitatively in Figure 9. The main reason for this situation is to be found in the *reusability* of hardware components (chips), which are the cheap and reliable building blocks of hardware systems, small and large. To date, software components with similar properties simply do not exist, and reusable software ‘chips’ are not commercially available. The effort to design and produce such software would be too large, and standardization is not pursued by software makers who keep customers captive with proprietary software and computer platforms. As a result, software production cannot keep pace with the hardware technology, a situation often recognized as symptomatic of a ‘software crisis’.

![Figure 9: The comparison of Hardware/Software productivity](image)

**Figure 9: The comparison of Hardware/Software productivity**

A concern in this work was to try and produce visualization software that could, intrinsically, evolve as fast and as cheaply as hardware. This led us to select an *Object Oriented Methodology (OOM)* for constructing software components, specifically the component parts of our *SV* and *CFD* software.

OOM is a fairly universal approach that can be applied to solve many types of complex problems. The goal of OOM is to reduce the system complexity by decomposing it in manageable components called *objects*. Experience has shown that solving problems in a piece-wise manner leads to better quality and easily scalable solutions. The system is ‘cut’ into component pieces represented by ‘objects’ that interact, through well-defined interfaces, by exchanging information through *messages*.

The interesting feature of OOM is that objects can be created and developed independently, even with no a priori knowledge of the application in which the objects will be used. The existence of an object is independent of any specific application. The concept can be illustrated considering the *Partial Differential Equation (PDE)*. The PDE exists as a mathematical concept which can be applied, for example, in ship propulsion design or turbo-machinery flow analyses. The key aspect here is that the mathematicians do not need to know anything about the application of the PDE to develop solutions to the PDE. Hence, by encapsulating the knowledge on the PDE in an object (software component), one makes it available to and reusable by any application or software that may require it. The benefits are clearly a reduction in the application development effort, shorter production times and higher application maintainability. The benefits stem from the fundamental property of *reusability* of the PDE object. Any improvement to the PDE object would immediately benefit all the applications that use it. An interesting consequence of having many reusable software objects would be that it would then make sense to get hardware designed to fit the available software (and not the reverse as is the case today). The principles of software reusability and portability have been fundamental to all the work described in this thesis.
Reusability is an intrinsic feature of all OO software and their efficient exploitation promotes the computer network to become a commercial market place, as the Internet, in which such general-purpose and specialized software components needs to be available, validated and marketed [30].

The OO approach has led to the emergence of Object Oriented Programming (OOP) with specialized OO programming languages -- such as Smalltalk[31], CLOS, Eiffel[32-34], Objective C [35], C++ [36], Java, C# and other derivatives -- which apply encapsulation and inheritance mechanisms to enhance software modularity and improve component reusability. It is important to stress that the highest benefit of OOM is obtained when OOM covers the full software life-cycle, from the requirements specification phase to the software delivery phase. When an application is created applying OOM, reusability in different development phases can be expected. First, the OOP brings in object-oriented libraries, which provide components validated in previously developed applications. Second, the software design of the previously modeled software could be reused through the established design patterns. Previously developed components may be reused for the new application which does not need to be designed from scratch, which is an obvious advantage. Improvements that could be brought to the existing, re-used objects would also improve the ‘older’ applications that use the same objects.

It is interesting to note that the OOP paradigm has shifted the emphasis of software design from algorithms to data (object, class) definitions [37-39]. The object-oriented approach can be summarized in three steps. The system is first decomposed into a number of objects that characterize the problem space. The properties of each object are then defined by a set of methods. Possibly, the commonality between objects is established through inheritance. Actions on these objects and access to encapsulated data can be done randomly rather than in a sequential order. Moreover, reusable and extensible class libraries can be created for general use. These are the features which make OOP very attractive for the development of software, in particular for interactive software.

It should be mentioned that OOM does not directly reduce the cost of software development; however, it markedly improves the quality of the code by assuring consistent object interfaces across different applications. Estimated software construction times are often incorrect. Time and resource allocation tend to be largely underestimated in software projects, not uncommonly by factors of 2 to 5, especially where innovative features are to be developed. Unfortunately, for the software construction planning we do not have an underlying engineering, scientific or mathematical model to calculate the software development time required, when starting the new software development process. The theoretical basis of how to best construct software does not exist. The ability to plan the project costs, schedule milestones, and diagnose risk is ultimately based on experience, and could be only valid for a very similar application done in the past and applying the same development environment.

<table>
<thead>
<tr>
<th>RESPONSIBILITY</th>
<th>LAYER</th>
<th>TASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEVELOPER</td>
<td>Application - Database</td>
<td>simulation data</td>
</tr>
<tr>
<td></td>
<td>Visualization System</td>
<td>graphics data</td>
</tr>
<tr>
<td>VENDOR</td>
<td>Graphics &amp; Windowing System</td>
<td>displayable data</td>
</tr>
<tr>
<td></td>
<td>Device Drivers</td>
<td>low-level graphics instructions</td>
</tr>
<tr>
<td></td>
<td>Operating System</td>
<td>select hardware unit</td>
</tr>
<tr>
<td></td>
<td>Hardware Platform</td>
<td>I/O devices, CPU and memory</td>
</tr>
</tbody>
</table>

*Table 1: Layered Software Architecture*
It is also important to ensure the production of **portable** code, i.e. code that can run without need of adaptation on computing platforms other than its ‘native’ platform. Porting -- adapting software to a computer system other than the one for which it was originally designed -- can be a tedious and costly process. Portability can be improved by adopting standards which various hardware/system platforms support. For example, one may adopt the OpenGL standard which is supported by graphic boards. This ensures that only a small kernel of code must be modified before recompilation for another hardware platform. This is why it is necessary to breakdown the software architecture in carefully-defined layers as seen in Table 1. Understanding the lower layers of software is of little or no interest to most scientists; these layers involve graphics languages, windowing systems, communication mechanisms and device-drivers. The system vendors have understood this and provide ‘graphic platform systems’ which implement the four layers below the visualization system layer. As depicted in Table 1, the upper two system layers are under the responsibility of the software developer; they include data extraction and enrichment. The visualization system maps numerical data onto graphical data. The rendering of graphical data is delegated to the graphics and windowing layer. In this way, porting of the software system to any hardware platform is possible without modifying the visualization application.

There is often a need for graphics hardware (graphics boards) that can enhance the performance of the most demanding 3D-rendering tasks requested from the graphics and windowing layer. The ‘vendors’ layers deliver a well-defined set of callable graphics routines -- examples: OpenGL, DirectX -- collectively called the **graphic engine**. Today’s graphics engines become increasingly powerful as software implementations are replaced by hardware implementations (see Figure 9). This is the case for a variety of UNIX platforms, with graphics engines implemented in specialized hardware such as: Silicon Graphics GL and OpenGL, HP Starbase, HP-GL, SUN XGL, Xlib and PEXlib, Microsoft GDI, Quick Draw and Postscript.

In the beginning of the 1990’s, CFView was ported onto most of these graphical engines.

A graphics engine typically processes floating-point input data to generate graphics. Example of graphics data models are lines and polygons. Hence one assumes that line drawing and polygon filling are functions provided by the graphics engine, and one needs not be concerned with developing low-level graphics routines. One can therefore focus on generating the data sets that are needed to ‘feed’ the graphics engine.
To develop the visualization software, our approach must be ‘multi-disciplinary’ in the sense that it puts together an application engineer and a computer specialist in order to develop different application layers, as shown in Figure 11. The software development environment needs to enable the evolution of the software under development and has to provide a framework for porting applications across different hardware/operating systems/windowing systems. Also, it has to simplify the process of the creation of interactive graphical applications, with enabling the application engineer to have under control the application software layer and hide the lower software layers of the system, as depicted in Figure 11. Thus, the object-oriented approach was selected to introduce the necessary abstraction levels, necessary for organizing the inherent complexity present in the development of the scientific visualization software.

**R&D projects history**

In 1988, we started developing the SV system that was to be named ‘CFView’ [40]. At the time, the basis of the Object Oriented approach [41] for developing SV systems was being established. The object-oriented programming language (OOPL) C++ was chosen, because of its high performance in number crunching and its capability to support OOP constructs.

CFView [21] was designed to work with structured and unstructured mesh types. CFView was found to be particularly valuable when treating the CFD simulations of the European Hypersonic Database [42] and validating the simulations against experimental data. CFView allows the simultaneous inspection of fluid flow simulations with structured and unstructured grids; as it uses data extraction algorithms such as plane-cutting, iso-surface and particle-tracing for both grid models, to uncover interesting features hidden in such large data sets. These routines require large computational resources and tend to limit the interactive feedback of CFView. In order to overcome this problem, a heterogeneous system named “Parallel CFView” was constructed which distributes the computing load over several processors and permits real-time, interactive visualization of CFD data. The development of Parallel CFView [43] was part of the EU-funded projects PAGEIN [44] and EEC/ESPRIT PASHA [45].

In the PASHA project, CFView visualization system was ported to MIMD and SIMD platforms for comparative benchmarking. The performance of the MIMD implementation was evaluated on two industrial test-cases provided by the EUROPORT-1 project. Both SIMD and MIMD parallel machines were used to provide massive back-end processing power. A key achievement of the Parallel CFView project turned out to be the interface for communication between machines with heterogeneous architectures. An analysis was carried out to compare the performance of such distributed computing environment consisting of both: the SIMD and MIMD implementations and the sequential CFView system. The results showed that the parallel implementation of the extraction algorithms speedup the performance of the visualization process by a factor of 10, which proved the viability of such visualization system configurations.

PAGEIN demonstrated more effective resource utilization through the support of distributed computing and collaborative engineering. The goal was to seamlessly integrate scarce human expertise in domains like, supercomputers, massively parallel computers and data servers. PAGEIN enabled the engineers to exploit the computing resources and reach the critical expertise mass without need for co-location. For the users PAGEIN has integrated on top of wide area networks, applications and tools for supercomputing, visualization and multimedia. It has demonstrated the viability of such combined technologies in CFD to be exploited on a European scale by aerospace actors.

The Live Code Learning Multimedia System (LCLMS) [46] was the IWT founded project, in which the database and network aspects for multimedia data sets were researched. This project strongly influenced the author’s own research towards distributed and collaborative environments for engineering applications based on low-cost PC-platforms. LCLMS provided the basic technology upon which QFView was designed.
The development of the Quantitative Flow Field Visualization (QFView) system [47] was made in the ESPRIT Project 28168 ALICE. QFView is a distributed software environment that integrates EFD and CFD data processing, including flow field mappings with flow field visualization. QFView was devised to support a unified treatment of data while providing for:

- the validation of results from experimental and numerical systems, and
- the archival and retrieval of data from unified (experimental and numerical) flow field database.

Based on proven Internet and World Wide Web (WWW) standard technologies, QFView provides an integrated information system for fluid dynamics researchers (see Figure 12). QFView is a web-based archival, analysis and visualization system, which enables the manipulation and extraction of data resulting from laboratory measurements or computational simulations. The system is suited for combining experimental and computational activities in a single operational context. This results in an increase of productivity, since the system facilitates for the exchange of information between investigators who conduct the same or similar simulations/experiments in different geographical locations, can be conducted in collaboration or independently.

The rapid progress in all facets of fluid dynamics research has made it essential that the research activities are conducted in close cooperation between experimentalists, numerical analysts and theoreticians. In the early stages of the CFD development the improvements of the process was so rapid that it was expected to eliminate the role of EFD all together. However, experience showed that in order to study new phenomena experiments are still the most economical approach. The CFD codes, ones validated against experimental data, are the most effective tool for the production of data to build a comprehensive flow database [48]. The strengths of various tools in EFD and CFD should be used judiciously to extract significant quantities required for problem solving as shown in Figure 13.
Three laboratory experiments were performed at the VUB in order to test and demonstrate the capability of QFView regarding web-based collaborative data access, namely:

1. The flow downstream of a bluff body in a double annular confined jet [49-51]
2. In-cylinder axi-symmetric flows [52]
3. Flow pattern of milk between 2 corrugated plates [53, 54]

The experiments generated large data sets -- measured and calculated values of physical quantities-- which were visualized and compared in order to support the validation procedure of the obtained results.

**The flow downstream of a bluff body in a double annular confined jet**

The cold flow in the cylindrical combustion chamber of a prototype industrial burner was investigated. Two concentric annular axial jets were used to approximate this complex flow field; the flow downstream of the bluff body was investigated over a range of Reynolds numbers using flow visualization, particle image velocimetry and laser Doppler velocimetry, see Figure 14.

---

**Figure 13: Use of EFD and CFD tools**

**Figure 14: Laminar flow at Re, = 60: DPIV of nearly axisymmetric flow, LSV of vortex shedding and CFD at various degrees of non-axisymmetric flow**
In-cylinder axisymmetric flows

The main objective was to get an insight into the structure of in-cylinder flow during gas exchange processes due to its capital influence on the combustion process and consequently on the efficiency of internal combustion engines. A simplified in-cylinder flow in axisymmetric geometries and the flow through the inlet valve for steady and unsteady conditions were investigated and experimental results represented with the QFView system. Two test rigs were used for flow visualization studies and PIV measurements of the cylinder equipped with a central valve:

1. for the steady flow through the inlet valve and
2. for in-cylinder transient flow (of water) in a mono-cylinder.
The influence of the speed and valve lift on the in-cylinder flow in the motored mono-cylinder was investigated while in the steady-state flow test rig; only the effect of the valve lift on the intake jet and stability of the tumble motion was retained. The similarities and differences in the two types of flows were analyzed in order to understand the relevance of the steady-state experimental results to model the unsteady flow in the motored cylinder. Figure 16 (a) shows an example of visualization of the flow pattern in the mono-cylinder at 90° CA after top dead centre (ATDC). The details of the flow structure were presented in terms of maps for average and instantaneous variables such as velocity (Figure 16 (b)), turbulent kinetic energy (Figure 17), vorticity (Figure 18), normal and shear strain rates. The main findings were:

- the integral length scales are the same for the investigated stationary and unsteady flows, a finding which supports the hypothesis of the isotropy of the small-scale structures
- the footprint of the flow is given by the root-mean square velocity and turbulent kinetic energy fields
- the flow is a complex collection of local jet and recirculation flows, shear and boundary layers

![Figure 17: Turbulent kinetic energy field at 90°ATDC at 5 rev/min, valve lift=10 mm](image)

![Figure 18: Average vorticity field at 90°ATDC at 5 rev/min, valve lift=10 mm](image)

**Flow pattern of milk between 2 corrugated plates**

A detailed calculation of the flow pattern of milk between 2 corrugated plates was carried out using 2D and 3D CFD calculations. The 2D calculation shows the influence of the corrugations shape, but the 3D calculations are necessary to assess the importance of the corrugation orientation. A model was constructed that allowed for a positive qualitative validation of the simulation results. The calculations allow identifying those regions where turbulent backflows and thus higher temperature regions near the wall can occur. These regions are the most sensitive to fouling and should be avoided as much as possible through better design. In this respect, CFD can be regarded as a valuable assistant for the design optimization of plate heat exchangers.
The development of the QFView environment highlighted the need for computer specialists and engineers to collaborate. Our work demonstrated the indispensable role of multi-disciplinary teamwork for progressing scientific visualization systems and developing next-generation engineering environments, which need to combine CFD codes and EFD experiments facilities in an integrated information framework, applicable to different sectors of industry, research and academia.

Figure 19: Experimental and CFD model for flow analysis between corrugated plates

Figure 20: Workflow for the integration of EFD and CFD simulations
After the September 11-2001 event, when the ALICE project ended, the author’s research shifted towards general-purpose (as opposed to scientific) visualization systems. This work was carried out in the LASCOT project [55] funded by IWT, being part of the European Information Technology European Advancement (ITEA) program for R&D of software middleware. The LASCOT visualization system demonstrated the possibilities of 3D graphics to support a collaborative distributed knowledge-management and decision-making applications. The research challenge was to deliver a visualization system capable of enhancing 'situation awareness’ (i.e. information that the actor-user has to manipulate to solve a crisis situation), which was done by providing a 3D interface that gave the user the possibility to navigate in and interrogate an information data-base in a highly intuitive manner. It was a main requirement that the visualization technology should be capable of assisting the user in performing decision-making and knowledge management tasks. The LASCOT’s visualization system was built upon the JOnAS [56] application server (developed on the European J2EE architecture). These technology and architecture could be easily used for developing new engineering integrated environments.

The work presently carried out by the author in the ITEA SERKET project is about security issues and protection against threats, which relates to the development of 3D graphical models capable of integrating and rendering data acquired from heterogeneous sensors like cameras, radars, etc.

Over the last 20 years, the author has initiated and worked on many research projects with a clear focus on developing scientific visualization software and advancing the state of the art in this area. There are potential avenues of future research, as discussed in the next chapter.

**Towards an Integrated Modeling Environment**

Today’s trend in software development is towards more intelligent, multi-disciplinary systems. Such systems are expected to capture engineering intelligence and to put in the hands of the engineer advanced tools for designing new products or performing investigations.

![Figure 21: Example of an Integrated Modeling Environment [57]](image-url)
The Integrated Modeling Environment (IME) [58] concept is quite recent, yet its roots can be found in 1st-generation CAD-CAM tools. An IME system attempts to offer to the engineer a homogeneous working environment with a single interface from which various simulation codes and data sets can be accessed and used. In the fluid mechanics application area, an IME system needs to integrate the latest CFD and EFD ‘good working practice’; the system must be constantly updated so that at any time, it runs on the most-recent software/hardware platform (see Figure 21).

An IME system consists of an Internet portal from which the investigator is able to access information/knowledge/databases and processing functions, at any time and wherever they are located/stored. He/she has access to accurate and efficient simulation services, for example to several CFD solvers. Calculations can be performed extremely fast and cheaply where solvers are implemented as parallel code, and grid computing resources are available. Results obtained can be compared with separate experimental results and other computations; this can be done efficiently by accessing databases that manage large collections of archived results. The possibilities for benchmarking and for exchanging knowledge and opinions between investigators are virtually infinite in an IME environment. Clearly though, a pre-requisite for an IME environment to work is its adoption by its user community, which agrees on a specific codex that enables and guarantees openness and collaboration. Typically, an IME system will open Web-access to:

- Computational Services: selection of simulation software and access to processing and storage resources
- Experimental Services: access to experimental databases with possibility to request new measurements
- Collaborative Services: chat and video-conferencing, with usage of shared viewers (3D interactive collaboration)

Visualization is required to support many tasks in an IME software. This poses the problem of building/selecting data models that can be used by the visualization components to present the information correctly to the users, whilst offering to them tools for real-time interaction in a natural, intuitive manner. The IME can include wall-displays connected to high-performance, networked computing resources. Such systems and architectures are no longer a mere vision: they are becoming reality, which opens new challenges for scientific visualization software researchers and developers.

Figure 22: Example of the 3D virtual car model testing
The usefulness of IME can be illustrated by considering how it would help the car designer. Assume that the present goal is to find out how a new car performs under various weather conditions and to assess its safety level before manufacturing begins. Assume that test-drive simulations are performed digitally, and that they are run in parallel. Clearly, car behavior problems can be readily identified, and design can be modified rapidly and at little cost; the expected savings in time and money are clearly substantial.

The QFView system that we have developed provides certain features of IME systems like the web-access and data sharing but does not integrate distributed computational resources, which might be addressed in the coming future.

**Thesis Organization**

The motivation for the research work presented in this thesis, and the objectives that were pursued are described in the Introduction Chapter, which also covers the state-of-the-art and an outline of the author’s research and development work in relation to object-oriented visualization software.

The main body of the thesis is then subdivided in 3 Chapters. The first Chapter is dedicated to discussing modeling concepts and fundamental algorithms, including discretization, i.e. the modeling of the continuum in cells and zones. This Chapter also presents the theoretical foundations of the basic algorithms applied to geometries, scalar and vector fields.

In Chapter 2, the visualization tools are explored to show different possibilities of extracting and displaying analyzed quantities. Improved and adapted visualization techniques are illustrated in several examples. Interactivity is then considered by explaining the Graphical User Interface design model. The Chapter ends with a discussion of Visualization Scenarios which allow the standardization of the visualization process.

Chapter 3 is devoted to discussing the relevance and appropriateness of the object-oriented methodology (OOM) for visualization systems, and the associated implementation issues. Object-oriented programming concepts and object-oriented data model are reviewed. The design and implementation of the visualization system is presented -- architecture, description of important system classes, management of input/output files, and development of portable and reusable GUI code.

The last Chapter explains how OOM permitted the development of “Parallel CFView”, an extension of the basic CFView visualization system that takes advantage of distributed and parallel processing. The Chapter covers the upgrading of the QFView system to distributed data storage/archiving for scientific applications, then its application for visualization in general-purpose information systems as prototyped in the “LASCOT” and “SERKET” ITEA projects. We conclude this Chapter with suggestions for future research.

During the course of the work presented in this thesis, our CFView visualization system has evolved from an academic laboratory prototype to a versatile and reliable visualization product providing a highly-interactive environment capable of supporting the most demanding fluid flow investigations. Researchers can reliably use and control CFView to extract, examine, probe and analyze the physics of flow fields, by simple commands through an intuitive graphical user interface.

One of the objectives of this work was to explore object-oriented programming and design a new development methodology for scientific visualization systems. This objective was achieved with the production of CFView, a visualization system fully implemented in C++ (one of the OOP languages) [36].
1 Modeling Concepts and Fundamental Algorithms

The Object Oriented Methodology (OOM) software modeling concepts were applied and further developed in order to establish the software model of the SV process. The fundamental concept in OOM is the “object”; it is the elementary ‘building block’ for mapping scientific and engineering concepts to their software equivalents. The object is an abstract construct, which approximates (in a simplified manner) the understanding of real concept in consideration, which is often quite complex. Consider, for example, how the physics of fluid flows are described in terms of numerical equations and how these equations are modeled by software objects. These objects are useful because they are identifiable elements with a well-defined purpose: each object performs a given function by encompassing a certain mathematical or physical ‘intelligence’ For example, an object modeling a second-order differential equation; or a object modeling the viscosity of a liquid at a given temperature; etc., in such a way that it can be ‘re-used’ by the software engineer with no need to understand the internal working details of the object. The obvious reused object in the real life is a car. We need it to go from one place to another, but we do not need to know how it is build. We use it, and this is a way the software engineer is supposed to reuse objects.

![Diagram of software model as communication media in the software development process](image)

The software model is a fundamental element in OOM software development. The model describes the knowledge mapped in the software in a formal, unambiguously defined manner. Such a precise specification is both the documentation and the communication tool between the developers and the users; recall that the term ‘developer’ includes application analysts, software designers and coding programmers (see Figure 23).

In the software development process, the analyst creates an abstract model that will be partially or fully implemented. The designer uses that model as a basis to add specific classes and attributes to be mapped onto one or more OOP languages. The designer specifies the detailed data structure and functional operations/processes, which are required by the application specification. Finally, the programmer receives the analyst’s and the designer’s models for implementation into source code. The source code is compiled to produce the executable software. Software modeling is then the iterative and incremental process which maps abstract concepts into formal constructs that eventually become reusable software entities. In OOM, the object model comprises a data model and a functional model; see section 3.2 Object Oriented Concepts. The specification of an object includes a description of its behavior and of the data necessary and sufficient to support its expected functionality. The data model describes the pertinent data structures, the relations between the objects and the constraints imposed on the objects. The functional model describes the objects’ behavior in terms of operations. From the data model point of view, the primary concern is to represent the structures of the data items that are important to the scientific visualization process and the associated relationships. The
modeling tool that was applied is known as the *Entity-Relationship Model (ERM)* [59]; it is well-adapted for constructing static data aspects and ensures the completeness of and the consistency between the specified data types. Figure 24 depicts an *ERM* with *Surface*, *Section* and *Vertex* entities (shown together with their relationships). Modeling entities define the direct association between problem entities (visualization data) and software objects (manipulative data), which establishes the reference basis for the incremental software development. This is important, because software modeling is not a one-shot process but a continuous or repetitive one, since software must be changed to account for the evolution of the user requirements and/or of the technology platforms.

It is important to mention that the terminology introduced by *OOM* standardizes the names of the objects which will constitute the system in all development phases, so that the *Naming Convention* must be strictly preserved and complied with in the software model. *ERM* consists of three major components:

1. *Entities* that represent a collection, a set or an object and which are shown as rectangular boxes (*Surface*, *Vertex*). They are uniquely identified, and may be described by one or more attributes.
2. *Relationships* that represent a set of connections or associations between entities; they are shown as diamond-shaped boxes.
3. *Attributes* that are properties attached to entities and relationships and which are shown in oval ‘call-outs’.

In Figure 24, the integer numbers on each side of a relationship-type denote the number of entities linked by the relationship. Two numbers specified on one side of a relationship indicate min-max values. For the ‘consist of’ relationship, the *Vertex* can be part of one and only one *Surface* and a *Surface* consists of *M* Vertices. A dependency constraint on a relationship is shown as an arrow; for example, a *Section* can exist only if there is a *Surface* for which an *Intersection* exists. The attribute can be of atomic or composite type. An example of relationship attributes is the *Intersection* relationship, which has additional data attribute to perform the intersection operation by interfacing *Surface* and *Section* entities. Composite attributes are shown in double-lined ovals. Underlined attributes are key attributes whose values are unique over all entities of that given type. If an entity type does not have a complete key attribute -- like the *Vertex* because two vertices constructing two different surfaces can have the same index --, it is called a ‘weak’ entity type and shown in a double-lined box.

The attribute *index* of a *Vertex* is a partial key only and is denoted by a dotted underline. Entity types can be related by a ‘is-a’ relationship, which define specialization or generalization of related entities. The entity types connected by a small diamond determine a hierarchy (or lattice). For example, possible surfaces can be **structured** or **unstructured**. All the attributes, relationships and constraints on an entity type are inherited by its subclass entity types.

Figure 24: Entity-Relationship Model
The entities in the ERM diagram are the basis for the Class decomposition because of their one-to-one correspondence with the classes in the OOM Class Diagram. OOM preserves the same class decomposition throughout the software development process; see Section 3.7, where detailed ERM diagram of the scientific visualization model is shown, applying the ERM modeling elements showing objects, relations and constraints. For example, Mesh or Surface objects are entities, qualified by specific attributes (for example color or coordinates). Their relation includes functional dependencies between them, while constraints are applied on attribute values.

ERM naturally fits in OOM, because it improves the specification of the object data model. ERM is one of the semantic data models that support a rich set of modeling constructs for representing the semantics of entities, their relationships, and constraints. The entity types are mapped to classes and their operations. Constraints that cannot be specified declaratively in the model are coded in class methods. Additional methods are identified in the classes to support queries and application functionality. ERM makes the software model more tangible to users who just have CFD application knowledge. It is advisable to use ERM as a communication tool between users and developers because ERM gets better the specification of the software model and its analysis.

The modeling concepts are not simply data-oriented: the object data model comes together with the fundamental algorithms which will generate the inputs to the computer graphics algorithms. Algorithms represent a suitable abstraction of the system’s behavioral characteristics; their modeling is done in conjunction with the data structures, and puts together all information required to make them work. The algorithmic models define the computational aspects of the system; these models can be analyzed, tested and validated independently of the full system’s implementation. The algorithmic solutions influence directly the performance of the implementation; hence, they should be modeled as simple, efficient and traceable components. To achieve clarity, effectiveness of expression, and conciseness in describing algorithms, we will use a ‘pseudo-language’ which can be routinely mapped to any formal high-level programming language. An algorithm is written as a sequential order of statements; a statement can be expressed in a natural-language syntax or in as mathematical notation with formal descriptors and list of variables. The important statements are:

- assignment: \( a = b; \)
- conditional: \( \text{if condition then statement else statement;} \)
- loop: \( \text{for variable = value until value do statement; while condition do statement} \)

In OOM, the data structures are encapsulated in the objects together with the algorithms. Normally, the algorithms operate just on the internal data structure of the object they are associated with. More complex algorithms could involve several objects; in this case, the data structure supporting their interaction needs to relate all of them. For example, the cutting-plane algorithm involves the geometrical data structure of the domain and of the surface which is created as a result of applying the algorithm. In what follows, we have grouped the algorithms according to the data structures involved in the visualization processes, i.e. in terms of: combinatorial topology (cell connectivity, node normal), computational geometry (cutting plane, section, local value) and quantity representations (iso-lines, iso-surfaces, thresholds and particle traces).

Computer graphics algorithms which perform operations such as coloring, rendering and shading are commonly implemented in hardware (and are of no interest here). What is important is the way in which we configure their set-up and inputs in order to create the desired representations (this is partially discussed in Chapter 2 Adaptation of Visualization Tools dealing with Representations).

This Chapter describes the algorithms in the following order: starting with topology, extending them to geometry and ends with quantity related algorithms. This order was chosen because it incrementally introduces new data structures which are sufficient and necessary for the algorithms to be operational. For example, the cutting-plane algorithm cannot be efficient without the topology information.
### 1.1 Data Modeling

*Data modeling* is an important step in the visualization process since the data to be visualized needs to be appropriately organized and structured. The *Data Model* has a direct impact on the performance of the visualization system, and this is especially important when one must handle and store vast amounts of persistent data. The data modeling definition is essential in supporting different visualization techniques (as explained in Chapter 2 Adaptation of Visualization Tools).

*CFD* is concerned with the finite set of objects defined using *discrete mathematics* concepts [60]. Such objects might be nodes, cells and similar types of objects describing the numerical simulation problem. For example, a grid line is defined by its starting and ending nodes and all the intermediate nodes; it is a ‘countable’ object since it is composed of a finite number of nodes. Other objects -- such as planes or lines -- are composed of infinite set of nodes as depicted in *continuum mathematics*. Discrete methods and combinatorial reasoning provide the theoretical basis for defining data structures and for analyzing algorithms. There is a tight relationship between mathematical objects and software objects; such identified relationships form the stable scientific background for software development and are clearly at the core of visualization system software design.

The geometric model in the Euclidian space [61], and its manifold model in the topological space [62, 63], are well covered by numerical models; we extended their application to provide the basis for creating graphical models for scientific visualization purposes.

Mathematically, a *set* is a collection of distinguishable objects that share some common property (or properties) that determine(s) their membership in the set. The objects in a set are called ‘members’. The *set of nodes* and the *set of cells* are the elementary sets required to discretize a continuum. The discretization process consists in dividing a spatial region into a finite number of non-overlapping cells, each of which is defined by its set of nodes. To correctly approximate space, we use a variety of cell types, each type being characterized by the interpolation function that is used to compute a physical quantity value at any arbitrary point within the cell. The approximation must be such that (physical) continuity is satisfied between adjacent cells. The interpolation algorithms work using the (physical) variable values (known) at the node coordinates. The *set of nodes* and the *set of cells* are modeled with the so-called ‘Zone object’.

*Figure 25:* Data model decomposition
The visualization data model covers the objects needed for storing and manipulating data; it includes two ‘decomposition’ lines:

- ‘vertical’ decomposition, based on sup-sub relationship, and
- ‘horizontal’ decomposition, based on part of relationship.

Both decomposition approaches describe the Cell and/or Zone geometry and topology with the grouping principle (see Figure 25). The vertical decomposition describes the sup-sub relationship as the principal concept in the boundary representation (‘B-rep’) model [64-66], which defines geometrical shapes using their limits. The sup-objects are here described by the group of sub-objects which are defined in the parametric space of lower dimension. The horizontal decomposition implies that the grouping principle puts together objects from the same parametric dimension, so that the resulting object remains in the same parametric space. The geometric model is fully described with the parametric and modeling coordinates, for example a point in a curve or surface is completely defined by its coordinates. The geometric model is enriched with the topology where boundary relationships and connectivity can define multiple regions. The multiple region objects with connected parts are defined by maintaining information about the common boundaries between them.

The Cell and Zone models are fundamental to the discretized geometry and topology models; according to the vertical and horizontal decomposition principle:

- the ‘cell’ is explained in the light of the vertex-based boundary model
- the ‘zone’ is terms of its horizontal decomposition of as a collection of cells.

### 1.1.1 Cell class

A ‘cell’ is the smallest identifiable element in/of a ‘zone’: it can be seen as the ‘minimal zone’, i.e. as a zone that consists of just one cell. Hence, the 3D-cell represents the smallest element of the 3D Euclidian space. In the same way as a 3D-cell defines the smallest 3-dimensional volume, the 2D-cell defines the smallest surface area in two dimensions, and 1D cell defines the smallest curve in the plane. Cells of parametric dimension equivalent to the zone cannot be decomposed further without introducing additional nodes. Hence the ‘cell’ is the simple-connected manifold of a modeled geometry [67] compose the curve, surface or space Zone. The ‘cell’ is defined in the modeling and in the parametric space where the mapping functions are known, as described in section 1.3.1 Isoparametric mapping and shape functions. These mapping functions are used to calculate the cell properties at any arbitrary cell point.

The cell concept is used in the Finite Element Method (FEM) (Zienkiewicz [68, 69]) and in Computational Fluid Dynamics (CFD) (Hirsch [60, 70]). The special types of elements, called ‘iso-parametric elements’, are applied in this thesis to support the Cell modeling. The name ‘iso-parametric’ derives from the fact that the same parametric functions describe the geometry and the quantity variations within the cell. With iso-parametric elements the variation of a quantity is interpolated using non-dimensional ‘iso-parametric’ coordinates. The cell’s parametric space is the region that fully describes the cell interior and boundary; it corresponds to the number of parameters required to define a point in it. For example, there is only one parameter needed to specify a point in a 1D-cell (as is the case for a point on a curve). The modeling dimension is the number of coordinates required to specify a point in the Euclidean space. If the cell description is dimension-independent, it allows geometry manipulation in the parametric space. An ‘iso-parametric mapping’ is applied only when precise coordinates inside the cell interior are required.

Parametric forms are modeled with ‘Map classes’ that support the mapping between the parametric and modeling spaces. The iso-parametric mapping is defined for each cell type (see Section 1.3.1). Maps are mathematical objects which satisfy smooth, continuous conditions, as a functional mapping of the form:

\[ M(p): D \Rightarrow I, \]

where \( D \) is the domain and \( I \) is the image.
where $D$ is the domain of map $M$ in the parametric space $R^p$, and $I$ is the image of $M$ in modeling space $R^m$ (where all $M(p), p \in D$ belong). The parametric dimension is defined by the cell itself as $1D$ for curves, $2D$ for surfaces and $3D$ for solids. Every object is embedded in a Euclidian space of some dimension called the modeling space. The cell map defines the underlying extent of the geometric object through a fixed number of parameters. Every underlying extent is represented as a parametric function, for example, a $3D$ surface:

$$
\begin{align*}
x &= X(u,v) \\
y &= Y(u,v) \\
z &= Z(u,v)
\end{align*}
$$

$x$, $y$, and $z$ in this map define the modeling coordinates and $u$ and $v$ define the coordinates in parametric space. The parametric dimension defines the extent of the cell and the modeling dimension is the dimension of the Euclidian space (as seen in the real world). Since the cell is a finite part of an infinite region, it does not consist of the entire curve or surface, but only of a trimmed region (see Figure 26).

The cell types may be distinguished by:

**TOPOLOGY:**
- number of nodes: $1, 2, \ldots N$
- skeleton: recursive structure of boundary cells

**GEOMETRY:**
- modeling space: node coordinates: $0D:(0), 1D:(x), 2D:(x,y)\text{ or } 3D:(x,y,z)$
- parametric space: node coordinates: $0D:(0), 1D:(u), 2D:(u,v)\text{ or } 3D:(u,v,w)$
- mapping function: transformations between modeling and parametric spaces
- interpolation function: inside modeling or parametric space

The cell is called $0D$, $1D$, $2D$ or $3D$ in accordance with the cell parametric dimension which describes the region of a point, curve, surface or body (see Figure 26 (b)). The cell boundaries are defined as cells of lower dimension in respect to the cell they bound (see Figure 26(a)): they delimit the cell region from the underlying Map. For example, three edges of parametric dimension $1D$ bound the face of topological dimension $2D$. Thus, $1D$-curve cells bound $2D$ surface cells. The edges are defined as regions with infinite line maps bounded by respective nodes.
A cell is defined by a set of nodes: it represents a bounded region of a geometrical space described by a boundary representation model (B-rep) model, defined in the indexing topology space. The same cell can be embedded in Euclidean spaces of different dimensions. A triangle is part of an infinite plane bounded by three lines; a tetrahedron is part of a 3D-space bounded by 4 planes. The same triangle could be embedded in the 3D space as a part of the tetrahedron B-rep. The definition of the lower-dimensional boundary cells (the ‘sub-cells’) provides an elegant recursive structure to describe a cell topology (see Figure 26), since boundary cells can in turn be defined by lower-dimensional boundary cells.

The cell classification identifies possible cell types (see Figure 26). A higher-dimension cell can always be described with cells one-dimension lower. This structure ends at 0-cell node. The sub-cell is a part of the boundary which defines higher-dimensional cells (named ‘sup-cells’). The sup-cell ‘knows’ which cells are its boundaries. For example, a tetrahedron is a 3D-cell bounded by four 2D-cells called ‘faces’. Each face is a 2D-cell bounded by tree edges. Each edge is a 1D-cell bounded by two nodes. Each node is a 0D-cell with no boundary.

The ‘composed of’ relationship is the main relationship used to decompose a cell in its vertex-based B-rep: it shows the cell identification path that goes from a cell of higher dimension (3D-cell) to a cell of lower dimension (0D-cell). The ‘part of’ relationship is the inverse relationship; it identifies the sub-cells which can define more than one sup-cell. For example, a ‘face’ is ‘composed’ of ‘edges’ and an edge can define 2 faces as part of the relationship. Both relationships define a collection of cells. In our case the composed of relationship is chosen for the implementation and the part of relationship is calculated from the composed of relationship, when necessary.

It is convenient, because of the variety of cell types, to group cells according to topology (number of cell nodes), geometry (parametric space) and interpolation functions. Topology permits to describe uniquely, for example, a 3D-grid as collection of nodes, edges, faces or 3D-cells. The thesis describes in details the 3D cells: hexahedrons, pentahedrons, pyramids and tetrhedrons and 2D cells (quadrilaterals and triangles). The definition of a cell is dimension-independent and is completely analogous for cells of higher or lower dimensions. For each cell type, the cell skeleton is defined and shown in appropriate cell tables (see Figure 27). The skeleton provides an easy way to traverse the topology of an object in a dimension-independent manner. A skeleton of specific dimension defines a set of cells. For example, consider the pyramid:

- the 0-skeleton are the five nodes,
- the 1-skeleton are the eight edges,
- the 2-skeleton are the five faces.
Figure 28: 1D & 2D Cell topologies

The applied indexing for uniquely identifying the cells topological properties is described in the following skeleton tables and related Figure 28 and Figure 29:

1D: segment

<table>
<thead>
<tr>
<th>Mesh type</th>
<th>unstructured &amp; structured</th>
<th>Cell topology</th>
<th>SEGMENT 1D, 2D &amp; 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Nodes</td>
<td>Axis Nodes</td>
<td>Coordinate System</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>T1N2</td>
<td></td>
</tr>
<tr>
<td>0-1</td>
<td>u</td>
<td>0-1</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: SEGMENT skeleton table

2D: triangle

<table>
<thead>
<tr>
<th>Mesh type</th>
<th>unstructured</th>
<th>Cell topology</th>
<th>TRIANGLE 2D &amp; 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges Nodes</td>
<td>Cell Nodes</td>
<td>Axes Nodes</td>
<td>Coordinate System</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>T2N3</td>
</tr>
<tr>
<td>0-1</td>
<td>0-1-2-3</td>
<td>u</td>
<td>0-1</td>
</tr>
<tr>
<td>1</td>
<td>1-2</td>
<td>v</td>
<td>0-3</td>
</tr>
<tr>
<td>2</td>
<td>2-0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: TRIANGLE skeleton table

2D: quadrilateral

<table>
<thead>
<tr>
<th>Mesh type</th>
<th>unstructured &amp; structured</th>
<th>Cell topology</th>
<th>QUADRILATERAL 2D &amp; 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges Nodes</td>
<td>Cell Nodes</td>
<td>Axes Nodes</td>
<td>Coordinate System</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>T2N4</td>
</tr>
<tr>
<td>0-1</td>
<td>0-1-2-3</td>
<td>u</td>
<td>0-1</td>
</tr>
<tr>
<td>1</td>
<td>1-2</td>
<td>v</td>
<td>0-3</td>
</tr>
<tr>
<td>2</td>
<td>2-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3-0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: QUADRILATERAL skeleton table
Figure 29: 3D Cell topologies
<table>
<thead>
<tr>
<th>Mesh type       : unstructured</th>
<th>Cell topology  : TETRAHEDRON 3D</th>
<th>Coordinate System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges 6</td>
<td>Nodes 4</td>
<td>Faces 4</td>
</tr>
<tr>
<td>0</td>
<td>0-1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1-2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2-0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0-3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1-3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2-3</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: TETRAHEDRON skeleton table

<table>
<thead>
<tr>
<th>Mesh type       : unstructured</th>
<th>Cell topology  : PYRAMID 3D</th>
<th>Coordinate System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges 7</td>
<td>Nodes 5</td>
<td>Faces 5</td>
</tr>
<tr>
<td>0</td>
<td>0-1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1-2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2-3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3-0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0-4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1-4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2-4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3-4</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: PYRAMID skeleton table

<table>
<thead>
<tr>
<th>Mesh type       : unstructured</th>
<th>Cell topology  : PENTAHEDRON 3D</th>
<th>Coordinate System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges 9</td>
<td>Nodes 6</td>
<td>Faces 5</td>
</tr>
<tr>
<td>0</td>
<td>0-1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1-2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2-0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0-3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1-4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2-5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3-4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4-5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5-3</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: PENTAHEDRON skeleton table
Mesh type: unstructured & structured
Cell topology: HEXAHEDRON 3D

Table 8: HEXAHEDRON skeleton table

<table>
<thead>
<tr>
<th>Edges</th>
<th>Nodes</th>
<th>Faces</th>
<th>Nodes</th>
<th>Cell</th>
<th>Nodes</th>
<th>Axis</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0-1</td>
<td>0</td>
<td>0-3-2-1</td>
<td>T3N8</td>
<td>0-1-2-3-4-5</td>
<td>u</td>
<td>0-1</td>
</tr>
<tr>
<td>1</td>
<td>1-2</td>
<td>1</td>
<td>3-0-4-7</td>
<td></td>
<td></td>
<td>v</td>
<td>0-3</td>
</tr>
<tr>
<td>2</td>
<td>2-3</td>
<td>2</td>
<td>0-1-5-4</td>
<td></td>
<td></td>
<td>w</td>
<td>0-4</td>
</tr>
<tr>
<td>3</td>
<td>3-0</td>
<td>3</td>
<td>1-2-6-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0-4</td>
<td>4</td>
<td>2-3-7-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1-5</td>
<td>5</td>
<td>4-5-6-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>7-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 30: Cell ERM
By convention, the local coordinate system of a cell is in parametric space: its origin is at the first cell node, and the coordinate axes are defined based following the local node indexing within the cell, as described under the Coordinate System header in each skeleton table. The orientation of the cell boundary is considered positive if its face normal points from the interior of the cell to the exterior. A cell is always embedded in a geometrical space of equal or higher dimension than the cell’s intrinsic dimension, and can be composed of cells of intrinsic dimension lesser than its own. The cell with lowest ‘0D’ dimension is a node. Any cell can be described by an ordered collection of nodes. The topology for nodes, edges, faces and cells is predefined for each cell type so that some properties are implicitly defined. For example, the order of the nodes of a triangular cell enables us to compute the line that is normal to that triangle in a 3D space. Another example is the extraction of faces from a 3D-cell applied in a marching-cell algorithm, see Section 1.4.1.

B-rep defines the lower dimensional cells as sub-cells, while the cell itself is defined (bounded) by the mentioned sub-cells. As shown in Figure 30, each cell can have more than one sup-cell (named ‘Stars’), and a cell’s boundary can be composed of more than one Boundary sub-cell. The sup-sub relationship is defined by the Bridge classes. The Bridge class defines the relationship between a cell and one of its boundaries; it specifies the direction, when moving from the boundary in the cell interior. Each boundary cell is a bridge; for example, a triangle has 3 bounding ‘bridges’ (edges) which determine the boundary sub-cells. Boundary cells can be connected in a prescribed way by showing the boundary topology, named Frame. The group of Bridges (edges) forms the Frame, which defines their sup-cell (face).

### 1.1.2 Zone class

The basic class that organizes and allows the manipulation of geometrical data is called the ‘Zone class’. A Zone is a geometric concept that defines a portion of the Euclidian space; it is defined as an ordered collection of nodes (also known in mathematics as a point set). A geometry described with the nodes topology defines curves, surfaces and bodies. Each node represents a unique set of real numbers, called coordinates, in the right-handed rectangular Cartesian coordinate system.

\[ P \text{ identical } (x,y,z) \text{ as } P(x_i), \quad i=1,2,3. \]

1-3

The geometry of a zone, described by the node coordinates and the topology is a composition of cells. The zone topology is defined as the union of cells according to the B-rep cell model extended to support:

- the intersection of every pair of the zone cells,
- the union of any number of the zone cells.

The finite space \( \Omega_z \) is discretized into a finite number of regions. The zone \( Z \) is expressed as the union of the finite space \( \Omega_z \) and its boundaries \( \Gamma_z \):

\[ Z = \Omega_z \cup \Gamma_z \]

1-4

Cell \( C \) is expressed as the union of the finite region \( \Omega_c \) and its boundary \( \Gamma_c \):

\[ C = \Omega_c \cup \Gamma_c \]

1-5

The boundaries of the cells represent their connections with the neighborhood cells:

\[ \Gamma_{\epsilon_1} \cap \Gamma_{\epsilon_2} = \Gamma_{\epsilon_1 \epsilon_2} \]

1-6

The common boundary \( \Gamma_{\epsilon_1 \epsilon_2} \) is also the zone cell. The zone is defined as a union of all its cells, whatever parametric space they belong to:

\[ Z = \Omega_z \cup \Gamma_z = \bigcup_{c=1}^N C_c, \quad \text{where } N \text{ is the total number of cells.} \]

1-7

The Zone model is defined with a boundary representation model, B-rep, because we always treat a finite region in the generalized modeling space. A zone encapsulates the relationship with its boundary. As the cell is the
atomic entity in the modeling and parametric spaces, it represents a Zone unit space. The only difference is that the Zone B-rep has in addition the cell connectivity to depict the zone topology. The zone topology incorporates schemes that specify the geometry of the zone boundaries and their inter-connectivity. It explicitly maintains cell connectivity by ensuring that adjacent cells share their common boundaries. The zone topology depends solely on the cells structure, not on their geometry. If the geometry of a zone is modified, its topological structure remains intact. The invariant nature of the zone topology allows its separate treatment (see Section 1.2). The combined use of geometry and topology is present in the algorithms described in Section 1.3 and Section 1.4.

Grids can be discretized using different types of zones, so one can establish sup-sub zone relationships and define a zone as a region bounded by zones of lower dimension -- the 'sub-zones'. A sup-zone is a zone bounded by sub-zones. The sup-sub relationship yields an elegant recursive structure called the zone topology. The B-rep modeling naturally supports aggregation, thus it can be extended to model the set of zones through composition based on the recursive sub-zone relationship. B-rep describes the topology of a zone modeled with all its boundary zones and their connectivity.

The presented topological structure and its general functionality is defined in the Geom class, thus applicable to Zone and ZoneGroup sub-classes, which are responsible for modeling the collection of zones as a Set. The additional topological classes are: Bridge, Frame and Skeleton.

Figure 31: The Zone Composition and Topology ERM
The *Bridge* class describes the relationship between a *zone* and one of its *boundary* zones. It specifies the orientation of the boundary so one can identify on which side of the boundary is the *interior* of the zone. It specifies not only that the sub-zone bounds the sup-zone, but also the *pre-image* of the sub-zone on the sup-zone domain. The pre-image map is defined as the map $M_p$ which satisfies:

$$M_{sup}(M_p(p)) = M_{sub}(p)$$

for all the parameters $p$ in the interior of the sub-zone, where $M_{sup}$ is the bridge sup-zone map and $M_{sub}$ the sub-zone map.

Pre-images relate points in sub-zones to parameter-values of the zone they bound. This functionality is notably used when moving to the interior of a surface from a point on its boundary. The inward direction from a boundary can be approximated by the planes tangent to the sub-zone and the sup-zone. This is not sufficient, however, since several sup-zones can use the same sub-zone as their boundary, and an imposed orientation is required to uniquely solve the problem. The sub-sup relationship imposes the *bridge orientation*, a Boolean variable which specifies the orientation of the sub-zone as ‘natural’ or ‘inverted’. The *bridge normal* is defined as the vector perpendicular to the sub-zone at the point considered and which lies in the plane tangent to the sup-zone at that point. There are 2 kinds of bridge normals:

1. The *algebraic normal* is computed from the natural orientation of the sub-zone and sup-zone.
   - *D0-bridge*: point bounds a curve, the normal is simply the oriented tangent of the curve.
   - *D1-bridge*: curve bounds a surface, the algebraic normal is defined as $n \otimes t$ where $t$ is the oriented tangent to the curve and $n$ is oriented normal to the surface.

2. The *topological normal* is the algebraic normal, but its orientation can be inverted if needed. It represents the inward direction from the sub-zone to sup-zone. The inward normal is closely related to the relative orientation of the bridge.

The *interior* of a zone is a region of the zone bounded by lower dimensional zones. A node $n$ of $Z$ is said to be an *interior* point of $Z$, if there exist at least one sub-zone about $n$ whose all points belong to $Z$. A node $n$ is said to be *exterior* to a zone $Z$, if none of the sub-zone points belongs to $Z$, in which the node is located.

The *boundary* of a zone is the zone of topological *dimension-1* composed of the nodes that are exterior to $Z$. A *Frame* combines bridges into a connected enclosure. Enclosures can have one *Outer* frame and several *Inner* ones (see Figure 33). The dimension of the *Frame* is that of the sub-zone whose boundaries it represents. *Frames* can be ‘active’ or ‘inactive’ depending on whether they have been assigned to the sup-zone or not. The *Geoms* can be constructed from *Frames* and *Bridges*. 
The **Frame** object represents the boundary as the set of Zones, which are connected and form a closed loop (see Figure 33 showing the case of a surface and a body with internal holes). **Bridge** and **Frame** are useful when the topological structure is manipulated in point location based algorithms. The **Skeleton** provides an easy way to traverse the topology of object with different parametric dimensions.

The **Range** object represents a rectangular box aligned with the coordinate axes and representing the minimum space holding the zone. The **Geom** hierarchy allows working with a single zone or with a set of zones. The **Zone**-operations apply the boundary topology for extraction algorithm, while the **ZoneGroup** operations are related to zone composition (assembly) and distributions of the underlying maps and topologies.

A set of cells defines a **Zone** when all the cells have disjoint interiors. A **Zone** is a grid of polygonal cells with tangent discontinuities at the cells boundaries. A **zone** is a topological entity which defines connectivity between **cells**. Each zone has a set of cells associated with it, together with a set of nodes from which cells are created. A **Cell** provides a piece-wise linear approximation and is always associated with a unique **Map**. Each cell is part of only one zone. The cell provides support for query/modification operations and can access its nodes. A **Zone** is defined only in the modeling space and it uses the cell maps for preserving the continuity of the geometry and quantity fields it is defining.

![Figure 33: Inner and Outer Frames of 2D and 3D zones](image)

**Figure 34: Zone-Cell ERM**
The Zone class models geometry reusing the cell definition in the modeling and parametric spaces. The point location algorithm, applies the marching-cell algorithm, which uses the zone topology, and ones inside the cell, the cell Map is applied to control the underlying cell extent in the parametric space. The cell boundaries split the parametric space in two parts, first the cell itself and second the rest of the Map. The rest of the cell Map is of no interest since it is not a part of the Zone. A Zone Map is a collection of independent Cell Maps.

The Node is the geometric element used to define the Zone points. A node defines a point in the Zone Map and also in the Cell Maps. The Cell Maps provide mechanisms to query the node coordinates and the derivatives of defined variables.

The cells connectivity is only defined for unstructured meshes. By definition, a zone is said to be connected if one can move between any two points in the zone along a curve that fully lies in the zone. As the interactively created Zones are unstructured, the zone connectivity needs to be generated dynamically during the interactive session.

Each zone subclass is specialized according to its intrinsic parametric cell dimension as 0D, 1D, 2D or 3D zone as the nodes are composing cells (see Figure 35, point, segment, polygon, polyhedral). The Zone class, which is
the base geometry class, is defined with different subclasses, which are each representing collections of homogeneous cells. The *PointSet*, *Curve*, *Surface* and *Body* are defined with the parametric dimension, respectively 0D, 1D, 2D and 3D. Recall that a zone is both a collection of nodes and a collection of cells. The highest parametric dimension of the cells in a zone determines the dimension of the zone and its parametric space. The *parametric dimension* of the zone (topology) should not be confused with the *modeling dimension* of the zone (geometry); the same holds for cells. For example, a surface with a 2D topology could have a 2D or 3D geometry definition, and its node coordinates would be \((x, y)\) or \((x, y, z)\) in the respective geometrical space.

As explained, different topologies can exist for the same geometry, and different geometries can exist for the same topology. Geometry and topology are defined in the *Zone* class.

### 1.1.3 Coherence and Tolerance Concept

A Zone can be either in a *coherent* or in a *non-coherent* state. The coherence status of a zone can be queried, and a non-coherent zone can be processed to become coherent. In this process, the map of the zone is modified so that its B-rep structure is coherent with the cell boundaries.

![Coherent and non-coherent cells](image)

**Figure 36: Coherent and non-coherent cells**

Zone coherence requires that the connected zones match along their all lower-dimensional boundary zones or not at all. For example, an example of not coherent curve is when it is self-intersecting (see Figure 36).

Coherence of 2D zones is achieved by combining triangular and rectangular cells. As cells do not have holes, they are simply-connected regions. Cells coherence guarantees that the Zone corresponds exactly to the trimmed map region, defined as the zone interior. This property assures that the connected surfaces meet across their common edges. As the result of the imposed coherence is that they must have the same nodes defining their common boundaries. The boundary representation imposes several geometrical constraints, for example, when an edge is constrained to lie completely on a surface. Such geometrical constraint is impossible to obey exactly since all computations are subject to round-off errors. Geometrical constraints can be met, only within certain tolerances, because the geometrical data are numerically approximated.

The tolerance-model has two essential parameters:

- the *problem size* that defines the coordinate units and
- the *unit tolerance* that specifies the distance between two points. Two points are considered one point if they are centered at the same origin when scaled down to the unit box of size 1.
Obviously, in practice, computation must be carried out at a given precision level, for example using double-precision floating point computation that yields an accuracy of about $10^{-16}$. The unit tolerance values are in accordance of a larger magnitude order than the precision limit given above, so that the inaccurate input and the accumulation of errors in the computation remain insignificant.

The effective tolerance for the problem is defined by multiplying problem size by input unit-size. Points are considered as coinciding when located within the effective tolerance distance. The cell has nodes that define modeling and parametric coordinates for points on the map. The cell nodes coordinates are created within specified tolerances.

### 1.1.4 Zone classification and Surface class

The data model must integrate all the objects necessary for storing and manipulating the data. This includes the Zone and Field classes that combine geometry, topology and quantity data for the objects created at input and for the objects created as the result of the user interaction. The zone classification follows:

- **0D** - general points:
  - local values,

- **1D** - general curves:
  - constant mesh line (for structured mesh),
  - arbitrary curve section,
  - general curve resulting from the intersection of two surfaces (for example, solid boundary and arbitrary section),
  - particle paths.

- **2D** - general surfaces:
  - mesh boundaries,
  - constant I, J or K surfaces (for structured meshes only),
  - arbitrary cutting planes,
  - isosurfaces,
  - surface-tangent vector (for example, stream-surface).

- **3D** - general volumes:
  - complete mesh,
  - single domain of the mesh,
  - arbitrary sub-mesh interactively generated.

In the next section, the 2D surface concept is explained in detail so as to clarify various aspects of interactively-created surfaces.

![Figure 37: Surface types: (a) open, (b) closed](image)
The analysis of a quantity field is usually performed on interactively-selected zones of different types which represent discretized regions of space. The most important *Zone* class is the *Surface* class; it is defined by a set of nodes and 2D cells. The surface’s finite geometry is in the 2D or 3D space. A surface lies in a bounded region of space and has two sides. To pass from one side to the other side of the surface, one must either cross the surface, or cross the curve that bounds the surface area. A surface without a boundary curve, (for example a sphere, or any surface that can be transformed continuously into a sphere) is called a *closed surface*, in opposition to and *open surface* which has at least one boundary curve (see Figure 37). The boundary curve is always a *closed curve*; if a boundary curve can continuously be deformed and shrunk to a point without leaving the surface space, then the surface is characterized as *simply-connected*.

![Figure 38: Simply and multiply connected and disconnected surface regions.](image)

The visualization system makes an extensive use of the surface as a main identification and manipulation object; most visual representations are related to surfaces, as the surfaces are the most appropriate containers of the data extracted from volume data sets.

The main surface types are:
- *mesh surfaces*
- *mesh boundary surfaces*
- *slicing or cutting planes*
- *isosurfaces*

The mesh surfaces are part of structured grids (consistent I, J, K surfaces or surfaces defined by faces), and other surfaces can be created for structured and unstructured meshes. The mesh and boundary surfaces are given at input, when cutting planes and isosurfaces are computed during an interactive session (surfaces and associated visualization tools are described in the Chapter 2).

The *cells connectivity* defines the relationship between nodes and various cells. A *Surface* can be queried for:
- which are the cells adjacent to a given cell,
- which are the cells that share a given node,
- which edges are defined with a given node,
- which two nodes define the edge connecting faces.

*Geometry algorithms* rely on the mathematical form of the underlying maps and on the topology that defines boundaries. They are:
- the *bounding cube* that computes the range (*cube*) which contains the geometric objects. It is used as rough bound of objects before proceeding to more precise computations,
- the *closest point* that computes the minimum (geometrical) distance between objects,
- the *extent* that computes the quantities as *length, area, volume* of objects,
- the *center of extent* that defines the *center of object* area, volume, etc.
1.2 Computational Topology

Computational (or algorithmic) topology is concerned with the development of efficient algorithms for solving topological problems. Computational topology is applied here to geometrical objects defined by point sets, [61, 63]; the resulting algorithms combine topology and algebra. The topological relationships are deduced simply by counting and indexing; these elementary techniques are sufficient to create the topologies of the constructed geometries, for example the polygons generation method applied in the cutting-plane algorithm uses the cell intersection pattern, which defines the cell topology, not requiring taking into account the coordinates of the cell nodes.

In this work, the topology algorithms are important for the unstructured meshes generation. For example, when a cutting-plane or iso-surface is generated, but not keep for further use, in that case it is sufficient creating a set of polygons and pass them to the graphics algorithm to produce the desired surface visualization (see Section 2.4.5). Thus, this process does not require the definition of the surface topology, but if such surface has to be kept for further analysis, then the polygon set representing the initial surface, becomes an input for the creation of the fully defined unstructured surface. The creation process of the unstructured surface requires the computation of the Zone topology consisting of:

- unique node and cell indexing
- cell topology
- cell connectivity, and
- unique local to global indexing map

This topology information enables different visualization tools of the SV system to operate on interactively-created surfaces. The first algorithm is the cell connectivity algorithm, which pre-processes cell connectivity information as part of the input to the visualization system. The second algorithm is the node topology algorithm, including the node reduction algorithm. This algorithm is used to determine surface normals which are required to identify the cells around a node and to calculate the unique node normal. This algorithm provides the necessary input to the surface-shading algorithm. Both algorithms are performing the nodes and cells topological mapping from local to global indexing space.

1.2.1 Topological space

The zone Z is a node-set describing a continuum concept. The topology C is a cell-set defined for a node-set Z having the following properties:

1. \( n \in Z \) and \( c \in C \)
2. the union of any number of cells of \( C \) is in \( C \)
3. the intersection of any two cells of \( C \) is in \( C \).

Figure 39: Zone and cell’s point classification
The pair \((Z, C)\) is called a topological space. In addition, each cell in \(C\) defines its topological space because it is defined as a node-set and as a cell-set containing its boundary cells. If the cell boundaries are excluded the open cell represents the open set of points which contains only interior points. The closed cell refers to a closed set of points and contains in addition its boundary points. A set is closed if the complement is open. All cell points that are bounded by its boundary are called cell interior. A point \(p\) is an interior point of \(c\), where \(c \in C\); if there exists an open set contained in \(c\) and at the same time contains \(p\). The set of all interior points of \(c\) is called the interior of \(c\). If the point is classified as interior point of all \(c\), it is in addition the interior point of cell-set \(C\) as \(c \in C\).

The cell interior is always a manifold because every interior point of the cell has its infinitesimal neighborhood which reflects the similar cell shape. The shapes of the curve, surface and body cell topology are shown in Figure 40, where the neighborhood of the interior point looks like the primary cell. Boundary points are not interior points or part of the manifold, because its neighborhood is not complete, see Figure 41. The concept of neighborhood is important in order to understand the definition of the boundary cell. The neighborhood \(\varepsilon\) is defined as infinity small region around the arbitrary point. It is a set of points inside the cell interior having the same parametric dimension as the analyzed cell. The neighborhood \(\varepsilon\) which completely belongs to the cell interior is defined as \(\varepsilon\)-full, and can topologically deform up to the analyzed point. The \(\varepsilon\)-full neighborhood classifies the point as the cell interior point.
Figure 42: Manifold and non-manifold neighborhoods

When the point is on the boundary which is not connected to another cell, the neighborhood is called $\epsilon$-half. The boundary sub-cell represents $\epsilon$-half connection to the sup-cell. The sup-cells interiors that are immediately adjacent to that point form the $\epsilon$-full neighborhood. For example, the boundary cell in such case represents an internal edge. The boundary cell retains non-manifold configuration. The non-manifold cells are cells in whose interior the shape of the original cell is perturbed, like shown in the Figure 42.

1.2.2 Structured and Unstructured Topology

The structured and unstructured topologies are the two forms through which the zone topological information is accessed. In this approach the zone topology is defined by unified concepts of:

- cell topology $\Rightarrow$ defines cell nodes
- cell connectivity $\Rightarrow$ defines adjoin cells

Figure 43: Zone topology concepts
Figure 44: Boundary LG map

The topological entities found in the zone topology are indexed cells and nodes. The cell topology contains the node indices defining topologically the zone as a set of cells. The cell connectivity contains the indices of cells connected to their cell sides, see Figure 43. The cell side represents the intersection between two cells. The cell which contains a non-connected side represents the cell on the zone boundary. The node and cell are uniquely defined inside a zone with three types of indices:

- global,
- local,
- cell.

The local indexing establishes the correspondence between the zone and cell nodes. The global indexing supports the node and cell identification between sup-sub zone topologies, which are usually applied in the cell based algorithms. The cell indexing enables the identification of the local node index inside the zone when the local cell index and the node cell index are known. The sub-sup relationship is the connectivity with external topological space. It is defined through local to global indexing, named LG map, in the case when the sup-zone is known. The LG map for nodes and cells defines global indices in the sup-zone, for nodes and cells respectively, see Figure 45. For example, if the zone is the surface partitions of its sup-zone boundary, the LG map with the internal bounded volume is defined, see Figure 44. Each zone topology is independent of its sup-zone, but occasionally, it is important to navigate through zone with different topological dimension in order to control the specific surface/volume region. This relationship is particularly important in the marching cell algorithms applied for extraction of section and cutting plane.

For structured grids the zone topology is given implicitly with the parameterized nodes order and all other sub-topologies can be deduced from such imposed nodes order, see Figure 45. This is not the case for unstructured grids where each node and cell index is explicitly numerically prescribed. In structured, topologies the nodes and the cells are identified with imposed number of parameters based on the parametric zone dimension. For structured zones the parameters are classified in Table 9.

<table>
<thead>
<tr>
<th>dimension</th>
<th>zone</th>
<th>parameters</th>
<th>number of nodes</th>
<th>number of cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>curve</td>
<td>( i )</td>
<td>1</td>
<td>1-1</td>
</tr>
<tr>
<td>2D</td>
<td>surface</td>
<td>( i, j )</td>
<td>I J</td>
<td>(I-1) (J-1)</td>
</tr>
<tr>
<td>3D</td>
<td>body</td>
<td>( i, j, k)</td>
<td>I J K</td>
<td>(I-1) (J-1) (K-1)</td>
</tr>
</tbody>
</table>

Table 9: Structured zone parameterization
Figure 45: The sup-sub zone relationship

(1) surface of the body, (2) curve of the surface, (3) node of the curve

In the Figure 45(a) the structured decomposition shows the possibility for the extraction of family of surface by making constant one of the $ijk$ indices. The procedure can be repeated for surface where the $i$- or $j$- family of curves can be extracted. Finally the node in the curve is identified with one index. The ordered node indexing makes possible the parametric manipulation of structured zones, see Figure 46.

For the unstructured grid the zone topology can be defined as homogeneous or heterogeneous set of cells. The simpler model is homogeneous zone topology which is constrained with only one cell type. For example the homogeneous zone topologies are surface of triangles or body of hexahedrons cell types. In the most general case the unstructured grid is composed of heterogeneous cells, which have varying number of nodes, edges and faces.

Figure 46: Node indexing in structured grids
Thus, the rule is to apply the vectors of integers for indexing cell nodes and connected cells controlled with additional vectors for cell indexing, see Figure 47. The internal structure of a cell is fully defined with the cell topology and the cell index vector. The same structure is applied to the cell connectivity with associated cell index vector. The cell index vectors vary in length. For the cell topology the cell index vector depends on the number of all the cell nodes while for the cell connectivity it depends on the number of all the cell sides. The cell index vectors identify the first cell node or connected cell, to the first cell side. In addition, the number of the cell nodes and its connected cells are defined from cell index vector. For heterogeneous unstructured zones the starting cell index for the cell topology CT and cell connectivity CC, are defined as:

\[
\text{cell topology } (\text{starting cell index})CT_i = \sum_{i=0}^{CN} n_i \\
\text{cell connectivity} (\text{starting cell index})CC_j = \sum_{j=0}^{CN} s_j
\]

where \( n \) is a number of nodes per cell, \( s \) is the number of sides per cell and \( CN \) is the total number of zone cells which precede the one for which the index is calculated. From the cell index vector the number of cell nodes are defined as the difference between the starting cell index CT of the next and current cell. The equivalent mechanism is applied to the cell index vector CC associated with the cell connectivity to define the number of cell sides, see Figure 47. Thus, the number of cell nodes and cell sides are calculated as:

\[
(\text{number of cell nodes})_i = CT_{i+1} - CT_i \\
(\text{number of cell sides})_i = CC_{i+1} - CC_i
\]

### 1.2.3 Cell Connectivity

The Cell Connectivity model defines the Zone topology consisting of:

- **Input model parameters:**
  - parametric dimension
  - number of nodes
  - number of cells
  - cell topology

- **Additional output parameters:**
  - cell connectivity
  - LG map for cells
  - LG map for nodes

The dynamic memory storage is appropriate for such heterogeneous data model as the size of the complete zone topology can be only defined if all the input topology is read in. The input data set are the first four parameters of
the Cell Connectivity model. The cell connectivity algorithm calculates the additional output parameters: cell connectivity and LG maps for the following tasks:

- to define zone topology,
- to define boundary topology,
- to define patches (segments) topology given as boundary conditions.

The cell connectivity information is required to make efficient the marching inside a zone when passing from one cell to another. This concept is applied in the cutting plane and vector line algorithms. The cell connectivity is calculated from the cell topology (nodes defining each cell) in order to keep the input data smaller and in addition, to avoid the unnecessary errors, which could be detected if the input data would contain inconsistent topology information. There are two types of cell connectivity calculations which are performed during visualization process:

- The first one is done during the preparation phase when the input files are generated. Such cell connectivity information is of the static nature for the zones input and doesn’t change during the visualization process.
- The second type of cell connectivity calculation is done when the surface is interactively created, and more precisely, when saved for additional investigation. These are cases when the cutting plane or the iso-surface is created, as the cell connectivity is generated for each newly created surface. This calculation is computationally intensive and affects real-time interaction. Such aspects impose further optimization and improvements to the cell connectivity calculation.

![Figure 48: Minimum node grouping rule](image)

**Initialization of empty sides set**

```python
for each cell:
    for each side in cell:
        if(side in sides)
            connect and remove side from sides
        else
            insert side in sides
```

**Metacode 1: The kernel of the connectivity algorithm**

The kernel of the cell connectivity algorithm is based on the grouping of cell boundaries, being they faces or edges, respectively for 2D or 3D topology. The cell connectivity algorithm receives the cell topology, which consists of nodes defining each cell. The grouping occurs around the minimum node index composing the side, see Figure 48, which is used as grouping key. This principle, which groups the sides around their minimum node index, improves the traversal of the cells. The cell connectivity algorithm gets better performance because the search is done just among the sides included in the node-side set. The supported side-types in 3D parametric space are predefined faces: triangles or quadrilateral, while in 2D they are edges as segments, see Figure 49.
Figure 49: Cell connectivity for different parametric zones
At the beginning each node-side set is empty. The algorithm iterates over all the zone cells and constructs the sides for each cell which are grouped according the minimum node index. When a side is created its minimum node index is found. The side nodes are ordered from the minimum to the maximum node index. By definition, the minimum node index is the first one in the array defining the side. This nodes order simplifies the comparison between two sides. As the node set is of fixed size and contains all the zone nodes, the appropriate data type to treat such set is the indexed array with the node index as entry. The node index makes possible to find the set of all processed sides around the identified node.

The newly created side is compared with all the sides in the node-side set. If such a side is not found, the created side is added to the node-side set. If there is a side in the node-side set, which matches the newly created side, the connected cells for the newly created side and the matching side are set respectively inside the cell connectivity table.

When discarding empty node-sides sets the resulting node set contains the node-sides sets, which contain non connected sides. Such sides represent the cells forming the boundary sub-zone. The sub-zone of parametric dimension-1 requires the complete definition of the zone topology. After the first traversal of all sides, the side nodes are grouped in the unique node set, which by definition doesn’t allow duplicate nodes. The sub-zone nodes, when sorted in the monotonous increasing order, represent the \( LG \) map of the boundary nodes. Simultaneously, the cell topology is created with the global node indexing of the sup-zone. Such sub-zone topology is completely constructed if a node and a cell indexing are done in the local index space. The related \( LG \) maps define the local indexing, which is applied to the cell topology and cell connectivity.

The result of the first iteration over boundary cells is:

- the \( LG \) map of nodes,
- the cell topology in global index space,
- the \( LG \) map of cells.

The following step is the calculation of:

- the cell topology in local index space,
- the cell connectivity.

The cell topology in local index space is created from the cell topology in global index space where for each global index the corresponding local index is found from the \( LG \) map of nodes. This algorithm is tuned with appropriated hashing value to improve the searching performance.

\[ \text{(a) surface bounds volume} \quad \text{(b) curve bounds surface} \]

\( \text{Figure 50: Closed boundaries} \)
The cell topology with the associated cell index array and parametric dimension of the cell types are sufficient parameters to find the cell connectivity of the investigated zone. When all the cells are connected, the remaining nodes which have additional sides represent the entry point for the definition of the zone boundary. The zone boundary can have one or more parts. Thus, connected or disconnected regions can be found. Each such region is defined as a closed zone. A curve bounding the surface or a surface bounding the body are examples of closed zone types, see Figure 50.

The cell connectivity algorithm is repeated on the boundary because different disconnected or multiple-connected regions could be found. A separate calculation is made imposing the test based on the front marching technique. In the collapsing algorithm, the necessary condition for the single-connected closed region is accomplished when the boundary results collapse back to the point, thus these are no boundaries of examined zone. If the collapsing is not accomplished the zone is an open zone with the associated boundary.

If the surface boundary exists it is always closed. For example, when surface has three boundaries it can form different surface parts arrangements, see Figure 52. It is obvious that it is not sufficient to prove that for the example shown in Figure 52 the surface boundary is the group of three curves. The question, if the zone is

```
unsigned int hash() const {
    unsigned int i=iv.length(), j=0;
    while(i--) j ^= iv(i);
    return j;
}
```

| Table 10: C++ implementation of the hashing value

Figure 51: Collapsing algorithm for multiple-connected and disconnected regions

Figure 52: Surface parts (a) multiple-connected, (b) disconnected and (c) mixed

Figure 53: Collapsing algorithm for multiple-connected and disconnected regions

The algorithm includes the traversal of all the sides on the front allowing the cell to be considered only once. Practically, the cell is added only once and removed only once from the front set of cells. The cell inclusion is controlled with the cell-done vector. When the front reaches the surface/curve boundary the remaining sides are not connected to any cell. The traversal of cells continues with the remaining ones which are not searched. The next cell becomes the starting cell for the creation of the new curve/surface zone. Such treatment splits the zone in multiple connected or disconnected regions, named ZoneParts, see Figure 51.

If the surface boundary exists it is always closed. For example, when surface has three boundaries it can form different surface parts arrangements, see Figure 52. It is obvious that it is not sufficient to prove that for the example shown in Figure 52 the surface boundary is the group of three curves. The question, if the zone is
multiple-connected or disconnected, still remains. The traversal of the cells starts with the arbitrary chosen cell and follows the cell connectivity information. The surface is recognized to be a connected region when all the cells are reached. In addition, if there are no boundaries the surface is closed. The algorithm is performed on surface/curve level as the created and visualized objects are in 2D/3D space.

When all the sides are traversed, the remained entries into the node-side set represent the boundary cells of the zone. These are, for a body the surfaces, and for a surface the curves. The boundaries are later used with the definition of boundary condition patches. For the internal and boundary surfaces the topology has to be defined from the global node index space. Thus, it is possible to reach sup-zones without leaving the zone itself from every cell inside the zone.

### 1.2.4 Node Topology

The node topology algorithm defines for each zone’s node the surrounding cells, which are sharing it. This information is very important for the conversion from the cell-centered numerical model to the vertex based one. Such conversion is performed for improving the surface rendering (cell normals to define a node normal) or when some derived quantities require cell values to be used to define the value at zone’s node. In order to fulfill the continuity condition for the calculated quantities, the node value has to be unique for all the involved cells, so an averaging mechanism, based on geometrical or physical properties, can be defined in order to calculate the unique node value. For example, the surface rendering requires a unique normal definition at each node. This is obviously not the case when each cell has its own normal and local mapping based on the vectors which define the cell edges. The only case where the normal is unique for all the cells is the case when all the cells belong to a unique tangential plane.

![Figure 53: Dual topology concept between nodes and cells](image)

The node-cell relationship has the $M<>M$ cardinality which implies that one node defines multiple cells, named node topology, and one cell is defined with multiple nodes, named cell topology. The dual principle of node and cell topology is shown in Figure 53. If we focus on the cell connectivity graph and we replace the cells with nodes at their gravity centers the result is the node connectivity graph. For both the concepts, the topological structure remains the same. Thus, it is generic to nodes or cells. In order to keep one data structure we recreate the dual structure when necessary. The topology and connectivity for cells are defined and the node dual structures are calculated when needed. The dual node topology structure is calculated in the case of the before mentioned averaging mechanism when the quantity is calculated from the values associated with surrounding cells. The node topology algorithm makes consistent the topological information in a zone. The achieved benefit is the reduction of the memory allocation. For structured zones, this information is implicitly given through structured indexing convention follows as trivial solution. For the unstructured case the algorithm is based on the
traversal of all the zone cells, where for each node the connected edges are found and surrounding cells are identified. The algorithm manipulates dynamic information which implies that the exact length of arrays storing topological information can be defined when complete cell/node traversal is performed. The required intermediate step is the creation of temporary sets for the processed state of nodes and the two disjoint sets storing surrounding cells. The two disjoint sets avoid recursive invocation of the cells which where previously processed.

Initialization of the node_processed state

Initialization of the node_topology set.

for each cell:
  for each node in cell:
    if(node_processed(node)) continue
    else
      node_processed(node) processed
      initialization of topology set
      initialization of cell_not_processed set
      insert cell in cell_not_processed set
      while(cell from cell_not_processed)
        insert cell in topology set
      for node find local_node in cell
        for cell find sides set containing local_node,
          for sides set find connected_cells set
            if(connected_cells not in topology set)
              insert connected_cells in cell_not_processed
              insert topology set in node_topology set.

Metacode 2: The node topology algorithm

The logic of the algorithm is outlined in the Metacode 2. The node_topology set has all the necessary information to recreate the node-centered topological information for a zone, see Figure 47 in Structured and Unstructured Topology section.

The node reduction algorithm is a minor modification of the node topology algorithm where the cell connectivity information is available and the unique node indexing has to be calculated for the given zone. The problem is illustrated in Figure 54. The elimination of duplicate nodes is based on coherence and tolerance model, see section 1.1.3. The nodes are assumed to be the same if they are within the range defined in the tolerance model. The local node identification is not based on the local index, as in the case of node topology algorithm, but on tolerance model. In addition, the new node index space is created.

Figure 54: Set of cells with duplicate nodes.
We assume that the connectivity information allows navigation around the node. The treatment of a particular case, when the node is building two cells, which are not connected through one of the cell sides, is resulting in having two nodes at the same place, see Figure 55. In such a case the boundary curve would intersect itself. This is avoided by applying node reduction algorithm, which eliminates the creation of non-manifold geometry.

![Figure 55: Navigation around the node: (a) one node (b) two nodes](image)

The second variation of node topology algorithm is the calculation of node value based on the averaging mechanism. The problem is to compute the cells surrounding the node and to calculate the node value from the cell values. The calculation of the node value is reusing the navigation part of the node topology algorithm in the simplified form for requested node. In addition, the cell can be customized to contain the information necessary to apply the desired averaging mechanism.
1.2.5 Domains Connectivity

In this section the connectivity information for structured non-overlapping grids is described. The domain connectivity is based on matching boundaries. It is shown how the orientation of the boundaries and segments is predefined together with rules which establish the matching between the two neighboring segments. In addition, the input description for the domain boundary conditions is given.

![Figure 56: Domain with boundary indexing in 2D & 3D and boundary identification](image)

The domain can be analyzed as a hexahedron cell type, see section 1.1.1, with the difference that the cell’s faces are domain boundaries, see Figure 56. The domain itself can be arbitrary oriented, but it must be right handed. Because of the structured nature of the domain node indexing \((i,j,k)\), the boundary indexing logic is different than face indexing of hexahedron cell. The rule defining the indexing is based on the permutation of the \(i,j,k\) planes in increased order for minimum index value and in decreased order for maximum index value. Applying such a rule the following Table 11 is defined:

<table>
<thead>
<tr>
<th>Boundary</th>
<th>Index</th>
<th>Value</th>
<th>Curve (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>(i)</td>
<td>1</td>
<td>((i))</td>
</tr>
<tr>
<td>B2</td>
<td>(j)</td>
<td>1</td>
<td>((j))</td>
</tr>
<tr>
<td>B3</td>
<td>(j)</td>
<td>max</td>
<td>((j))</td>
</tr>
<tr>
<td>B4</td>
<td>(i)</td>
<td>max</td>
<td>((i))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boundary</th>
<th>Index</th>
<th>Value</th>
<th>Surface indices(i,j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>(i)</td>
<td>1</td>
<td>((j,k))</td>
</tr>
<tr>
<td>B2</td>
<td>(j)</td>
<td>1</td>
<td>((i,k))</td>
</tr>
<tr>
<td>B3</td>
<td>(k)</td>
<td>1</td>
<td>((i,j))</td>
</tr>
<tr>
<td>B4</td>
<td>(k)</td>
<td>max</td>
<td>((i,j))</td>
</tr>
<tr>
<td>B5</td>
<td>(j)</td>
<td>max</td>
<td>((i,k))</td>
</tr>
<tr>
<td>B6</td>
<td>(i)</td>
<td>max</td>
<td>((j,k))</td>
</tr>
</tbody>
</table>

Table 11: Boundary indexing for 2D and 3D structured grids

This indexing logic has an interesting feature because it simulates the playing dice where the sum of indices for opposite boundaries is constant and equal 7. Note that it doesn’t matter whether the boundary is viewed from the outside or inside of the domain. The rule is that the lower index varies first and thereby the orientation is uniquely determined. As the boundary can be constructed with different boundary condition, additional decomposition of boundaries in segments (patches) is given. The segment is a part of the boundary with minimum and maximum extension.
In the 2D example, see Figure 57, there are 3 domains, each having 2 connected segments. For the domain $D1$, the involved boundaries are $B4$ and $B3$. On the boundary $B4$ there are two segments and the second one is connected to domain $B2$. Such identification path though $domain \rightarrow boundary \rightarrow segment$ is the origin for the following adopted notation. The unified segment identification as $Dd.Bh.Ss$, where lower case characters $d, b, s$ are replaced with numbers for specific case. Coming back to the example, there are three connected regions $I, II$ and $III$. The connectivity between them is specified in the following Table 12:

<table>
<thead>
<tr>
<th>Connected region</th>
<th>Segment</th>
<th>Segment</th>
<th>Orientation</th>
</tr>
</thead>
</table>

Table 12: Domain connectivity specification in 2D

There are eight possible orientations of the neighboring segments in 3D defined with reference node and orientation, see Figure 58. The first four possible cases of neighboring segment can exist if the origin of the segment is located in one of the four nodes as indicated in Figure 58, where the orientation between the connected segments is equal. The next four cases define the inverse orientation.

Figure 58: Domain connectivity in 2D

Figure 58: Segment orientation cases
In the following 3D example, see Figure 59, the 3 connected surfaces areas are identified. The segment orientation and the node reference are added to the 3D input. Applying the same symbolic path for the segment identification as the one for 2D, the following Table 13 is constructed:

<table>
<thead>
<tr>
<th>Connected region</th>
<th>Segment</th>
<th>Segment</th>
<th>Node</th>
<th>Orientation</th>
</tr>
</thead>
</table>

Table 13: Domain connectivity specification in 3D

The connectivity for the non-matching structured boundaries is removing one-to-one comparison of their nodes, and each passing from one boundary to the connected one is implying additional algebraic treatment of the point location [71].
Figure 59: Multi-domain (multi-block) connectivity in 3D
1.2.6 Cell Intersection Pattern

The cell intersection pattern is based on the cell model, see section Cell class. All components of cell topology are simply-connected and convex regions. The 3D cells are bounded with oriented faces which are themselves bounded with oriented edges. In order to define the cell intersection pattern the cell topology is modeled with the Winged Edge Boundary (WEB) model [72-74]. The WEB model supports explicit identification of nodes and faces starting from edges. ERM for 3D cell topology, see Figure 60, shows that an edge has the sup/sub topological relationships with other cells. The edge is composing a face (sup-relationship) and in addition it is constructed from two nodes (sub-relationship).

![Cell topology ERM](image)

Figure 60: Cell topology ERM

The important difference between the B-rep implementations are the choices which define the relationships stored explicitly in the WEB structure and the relationships left implicit (to be computed as needed). As mentioned previously in this chapter, the main B-rep topological items are: faces, edges and vertices. A face is a bounded portion of a surface; an edge is a bounded piece of a curve and a vertex lies at a point. Other elements are a set of connected faces, a loop of edges bounding a face and a loop-edge relationship (WEB) which are used to create the edge loops. In the full developed form the node model has three main relationships: edge <-> node, face <-> node and solid <-> node, see the skeleton tables 1-4 to 1-7. Such model is mapped to the WEB model in order to have a form in which an edge identifies its sub/sup cell, respectively node/face, see Figure 60.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Node A</th>
<th>Face A</th>
<th>Edge A</th>
<th>Edge B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>End</td>
<td>A</td>
<td>B</td>
<td>Next</td>
</tr>
<tr>
<td>(e^i)</td>
<td>(n_{start})</td>
<td>(n_{end})</td>
<td>(f_A)</td>
<td>(f_B)</td>
</tr>
</tbody>
</table>

Table 14: The WEB model record

Each WEB relationship requires a labeled record, see Table 14. In the cell topology section different B-reps are presented within the cells skeleton tables. These tables are defined with the vertex B-rep model, where edges and faces are defined with ordered set of nodes. Nodes order preserves the edge/face orientation, namely counter clockwise as seen from the outside of the cell, see Figure 61.
In the ER model of cell topology both node and edge models are present, see Figure 60. The WEB model extends the edge-based boundary model by the connectivity information between edges composing the face. Each edge $e$ appears in exactly two faces. The other two edges $e^A$ and $e^B$ appear for each edge $e$ in the faces description, see Table 14. Moreover, each edge has the consistent orientation of faces, where $e$ occurs exactly once in its positive orientation and exactly once in the opposite orientation. The WEB data structure takes into account the mentioned properties and identifies edges and faces based on the looping forced by the face orientation which can be clockwise or counterclockwise, see Figure 61. The face is defined with the starting edge and the face orientation. In addition, WEB includes the neighboring faces and the previous and next edge of the faces. As the full WEB data structure includes the neighboring edges, the edges for each node can be extracted.

The Figure 62 shows the WEB representation of the edge $e_5$. The face $f_2$ and $f_3$ are considered as faces $f_a$ and $f_b$ respectively. For example, the complete edge-based WEB model is constructed for the hexahedron; see Table 8, where the first node and face for each edge are also given.
The global edge orientation is given as direction from its start to its end node. This information is available from node-based B-rep, where edges are defined in terms of nodes. In Table 15, the face \( f_3 \) of the tetrahedron is defined with the edges \( e_1, e_5 \) and \( e_4 \). The face orientation determines the local orientation of edges, when defining the face itself. The WEB representation of the face \( f_3 \) is based on the global edge orientation, which is aligned with counter-clockwise orientation.

<table>
<thead>
<tr>
<th>First Node- Edge-Face</th>
<th>First Edge-Face</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node</strong></td>
<td><strong>Edge</strong></td>
</tr>
<tr>
<td>( n_0 )</td>
<td>(+e_0)</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>(+e_1)</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>(+e_2)</td>
</tr>
<tr>
<td>( n_3 )</td>
<td>(-e_3)</td>
</tr>
</tbody>
</table>

Table 15: WEB model of the tetrahedron

Such data structure treats faces and vertices in a completely symmetric manner. If the start node \( n_s \), end node \( n_e \), and the connected faces \( f_A \) and \( f_B \) are exchanged, we end up with the dual cell of the original one, see Figure 65.

The intersection cell pattern is based on cell nodes states which for each node can have a label set to \( TRUE(+) \) or \( FALSE(-) \). For each type of intersection pattern the node states are grouped together forming the node-mask. The node-mask captures the intersection pattern and it is mapped to a binary representation of an integer. The number of intersection patterns depends on the number of cell nodes describe with the following equation:

\[
\text{number of patterns} = 2^{\text{number of cell nodes}}
\]

The lookup table, contains all the possible intersection patterns for a given cell type. The unique integer label is created from the node mask, where each bit stores the node threshold state. The threshold is calculated for extraction algorithms like cutting plane and iso-surface are. For example, in the cutting plane algorithm the threshold is its distance sign from the plane and for the isosurface the threshold is difference between the scalar value and the given iso-value. Except for the computed node mask, based on geometry or quantity, and for the calculation of the intersection point or iso-value position, the marching cube algorithm is used unmodified for both: cutting plane and iso-surface algorithms [75].
Table 16: The lookup table for the tetrahedron

<table>
<thead>
<tr>
<th>Label</th>
<th>Node Mask</th>
<th>n</th>
<th>Edges Intersected</th>
<th>Faces Connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1 0</td>
<td>3</td>
<td>0-3-2</td>
<td>1-0-2</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0 0</td>
<td>3</td>
<td>0-1-4</td>
<td>2-3-1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1 0</td>
<td>4</td>
<td>1-4-3-2</td>
<td>3-1-0-2</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0 0</td>
<td>3</td>
<td>1-2-5</td>
<td>2-0-3</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1 0</td>
<td>4</td>
<td>0-3-5-1</td>
<td>1-0-3-2</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0 0</td>
<td>4</td>
<td>0-2-5-4</td>
<td>2-0-3-1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1 0</td>
<td>3</td>
<td>3-5-4</td>
<td>0-3-1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0 0</td>
<td>3</td>
<td>3-4-5</td>
<td>1-3-0</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1 0</td>
<td>4</td>
<td>0-4-5-2</td>
<td>1-3-0-2</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0 0</td>
<td>4</td>
<td>0-1-5-3</td>
<td>2-3-0-1</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1 0</td>
<td>3</td>
<td>1-5-2</td>
<td>3-0-2</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0 0</td>
<td>4</td>
<td>1-2-3-4</td>
<td>2-0-1-3</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1 0</td>
<td>3</td>
<td>0-4-1</td>
<td>1-3-2</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0 0</td>
<td>3</td>
<td>0-2-3</td>
<td>2-0-1</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1 0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 64: Intersection pattern and polygon orientation

The header of the lookup table, see Table 16, consists of the intersection pattern label, the node mask and the ordered sets of intersected edges and faces. The implied order of intersected edges guarantees that the scattered points of the intersection pattern define a polygon and that its orientation is in accordance with the global orientation of a created surface. Different orientations for the same intersection geometry have different node masks, see Figure 64.
The lookup table for each intersection pattern identifies which edges are intersected together with the intersected edge sequence, which preserves the polygon orientation. The edge is intersected, if its nodes have different labels. For each node labeled \textit{FALSE}, the traversal of the joint edges is done in the counter clockwise direction and for the node labeled \textit{TRUE} in the clockwise direction, as shown in Figure 66. This orientation is valid if the node is viewed from the cell interior.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure65.png}
\caption{Dual Cells}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure66.png}
\caption{Edge traversal direction for nodes labeled as: (a) FALSE (b) TRUE}
\end{figure}
When the navigation graph is coupled with node mask, see Figure 67, the unique traversal path is defined with the condition, where the looping is done around \textit{FALSE} nodes in clockwise direction. The intersection pattern identifies which edges are intersected. The edge ordering preserves the pattern orientation. The polygonal nature of the intersection pattern imposes looping over intersected edges in the predefined order imposed by the node mask. The algorithm takes advantage of the consistent edge/face orientation, see Metacode 3.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{tied.png}
\caption{The navigation graph for tetrahedron}
\end{figure}

If we consider the other cell types the number of intersection patterns increase as shown:

\begin{align*}
2^4 &= 16 \\
2^5 &= 32 \\
2^6 &= 64 \\
2^8 &= 256
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{pyramid.png}
\caption{The navigation graph for pyramid}
\end{figure}

\begin{center}
\textbf{Metacode 3: The cell intersection pattern algorithm}
\end{center}

- identification of node mask
- traversal of the mask
- node identification and its first intersected edge
- the edge becomes starting edge of the traversal.
- looping over intersected edges following the false node and face orientation
- the looping ends on starting edge
Figure 69: The navigation graph for pentahedron

For the reason of readability the example of the pentahedron lookup tables is given in appendix on page 231

The navigation graph for other cell types follows; see Figure 68, Figure 69 and Figure 70.

Figure 70: The navigation graph for hexahedron

The cell intersection pattern doesn’t imply that the created region is simply-connected. It can happen that we have up to four disconnected regions. The next complication is that polygons with greater number of nodes than triangles or quadrilaterals can compose the intersection pattern see Figure 71. In such case it is important to preserve the internal cell connectivity necessary for the creation of the zone topology.
The underlying cell map for the general n-polygon cell type is not defined. An unstructured surface can be composed only of triangles or quadrilaterals because they are the only cell types defined for 2D topologies. In order to subdivide the n-polygon cell type a very simple logic of subdivision is applied. It minimizes the number of created cells by generating quadrilaterals whenever possible. To eliminate random approach to the local subdivision (geometry orientation and connectivity) the Table 17 was defined as one possibility to uniquely define the polygon partitions.

<table>
<thead>
<tr>
<th>N-polygon</th>
<th>Subdivision</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>triangle</td>
</tr>
<tr>
<td>4</td>
<td>quadrilateral</td>
</tr>
<tr>
<td>5</td>
<td>quadrilateral + triangle</td>
</tr>
<tr>
<td>6</td>
<td>2 x quadrilateral</td>
</tr>
<tr>
<td>7</td>
<td>2 x quadrilateral + triangle</td>
</tr>
</tbody>
</table>

Table 17: Polygon Subdivision

The number of created internal cells depends on the number of polygon nodes. The unique triangles and quadrilaterals and their orientation (intersected edges order) are defined in the cases with more than four intersected edges. The practice shows that the maximum polygon has 7-nodes. This happens in the case of hexahedron cell. The subdivision generates one triangle and two quadrilaterals as shown in Figure 72. The advantage is that such decomposition produces less number of cells than the triangulated surface. The triangulation algorithm was made for the first version of marching cell algorithm.
In the intersection pattern algorithm the orientation of each internal polygon cell has to be preserved within all created cells. The connected faces are used in the marching cell algorithm, which can be combined with the cell connectivity algorithm, when the proper topological surface has to be created.

Table 18: Records from hexahedron lookup table with polygon partitions and multiple connected regions

<table>
<thead>
<tr>
<th>Label</th>
<th>nT</th>
<th>Triangles</th>
<th>Triangles</th>
<th>nQ</th>
<th>Quadrilaterals</th>
<th>Quadrilaterals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Edges</td>
<td>Faces</td>
<td></td>
<td>Edges</td>
<td>Faces</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0-4-3</td>
<td>2-1-0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>65</td>
<td>2</td>
<td>0-4-3 6-10-9</td>
<td>2-1-0 4-5-3</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>173</td>
<td>3</td>
<td>0-5-1 4-11-8 6-9-10</td>
<td>2-3-0 1-5-2 3-5-4</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>165</td>
<td>4</td>
<td>0-5-1 4-11-8 6-2-7 6-9-10</td>
<td>2-3-0 1-5-2 0-4-1 3-5-4</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1-4-3-2</td>
<td>3-2-1-0</td>
</tr>
<tr>
<td>170</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>0-3-11-8 1-9-10-2</td>
<td>0-1-5-2 3-5-4-0</td>
</tr>
<tr>
<td>188</td>
<td>1</td>
<td>6-9-10</td>
<td>3-5-4</td>
<td>1</td>
<td>1-3-4-5</td>
<td>0-1-2-3</td>
</tr>
<tr>
<td>225</td>
<td>2</td>
<td>0-7-3 4-11-8</td>
<td>6-1-0 1-5-2</td>
<td>1</td>
<td>0-5-6-7</td>
<td>2-3-4-6</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0-10-8</td>
<td>6-5-2</td>
<td>1</td>
<td>0-3-7-10</td>
<td>0-1-4-6</td>
</tr>
<tr>
<td>77</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>0-4-7-10 0-10-9-1</td>
<td>2-1-4-6 6-5-3-0</td>
</tr>
<tr>
<td>52</td>
<td>1</td>
<td>1-4-5</td>
<td>7-2-3</td>
<td>2</td>
<td>1-2-6-9 1-9-11-4</td>
<td>0-4-3-6 6-5-1-7</td>
</tr>
</tbody>
</table>

The new lookup table consists of the header indicating: intersection label, the created triangles/quadrilaterals and the connected faces for the marching cube algorithm. Obviously, the internal subdivision of polygon into possible quadrilaterals and triangles requires the cell connectivity indexing between internal cells. In the hexahedral example, see Table 18, there are defined faces which doesn’t exist for hexahedral cell. This faces are used for the internal cell connectivity indexing. It is assumed that any face with greater index then maximum index of the intersected cell face is an internal polygon cell.
The intersection faces are never connected internally and externally. Such possibility would setup non-manifold configurations, as shown in Figure 73, where an edge or a point would be connected to more than two segments or faces.

The intersection pattern algorithm treats only topologically the intersected cell. Such method only guarantees to produce sensible results when convex polygons are created. The pathological case, see Figure 74, shows the star-shaped polygon, which could happen if the algorithm would generate overlapping cells. However, such cases are not expected because the good numerical simulation grids are made from convex cells.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Bit representation</th>
<th>Edges intersected</th>
<th>Polygon created</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>F</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>$e_1, e_3$</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>T</td>
<td>$e_1, e_2$</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>$e_2, e_3$</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>F</td>
<td>$e_2, e_3$</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>F</td>
<td>$e_1, e_2$</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>T</td>
<td>$e_1, e_3$</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>T</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 19: Lookup table for the triangle
<table>
<thead>
<tr>
<th>Integer</th>
<th>Bit representation</th>
<th>Edges intersected</th>
<th>Polygon created</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>12</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>13</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>14</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>15</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Table 20: Lookup table for the quadrilateral

The only ambiguity that arises with is that for the intersection points, Figure 73, there are different possible connection patterns:

- connect the point on the next edge, clockwise or counter wise around the cell;
- surround regions of higher or lower-valued scalar quality;
- connect the opposite edges;
- calculate the average of all the four nodes, which will indicate the connection of the isolines.

It must be noted that the presented cell intersection pattern algorithm treats separately all the possible intersection cases for the defined cell types, without implying any symmetry or rotational similarity, in order to reduce their number. In case of ambiguity, when several connecting possibilities might exist, the one aligned with the imposed edge traversal is applied, see Figure 66. The presented algorithm generates unambiguous zone topology, only considering the cell intersection patterns, which differs from existing algorithms, which treat the influence of the quantity value change to remove ambiguities [76].
1.3 Continuity Assumption

The primary purpose of the scientific visualization system in Fluid Dynamics is to visualize the computed discrete properties, which describe the fluid flow as continuous matter. The fluid flow description is discretized throughout the numerical data model. The adopted Eulerian standpoint, as opposite to the Lagrangian coordinate system, which follows matter as it moves, describes the fluid quantities at specified points:

\[ Q = f(x, y, z, t) \]

where \( Q \) may be, for example, stress, density or velocity.

The continuity assumption describes the fluid fields: scalars, vectors or tensors, in the sense that the average quantity \( Q \) in a fluid surrounding the point are assigned to the point itself. This assumption ignores the molecular nature of the matter and the discontinuities which arise from this assumption. In addition, the motion of the molecules is neglected. It is assumed that the fluid is a continuous medium and its quantities change continuously. It means that the value of a quantity \( Q \) at point \( x \) and its value in the surrounding \( x + \Delta x \) can be related to its value at \( x \) by a Taylor series

\[
Q(x + \Delta x) = Q(x) + \frac{\partial Q}{\partial x} \Delta x + \frac{\partial^2 Q}{\partial x^2} (\Delta x)^2/2! + ...
\]

The first two terms are only required to indicate the behavior when \( \Delta x \) approach zero. Thus, one has a fictitious continuum supported by mathematical techniques, especially interpolation explained in the following section map the numerically generated data as discrete model in the continuum model.

In this section, the focus is on the development of techniques necessary to support the continuity assumption using numerically generated data as field quantities on defined geometry. These data represent the approximation of the given continuum by the set of points defined in the discretized space, called computational grid. The assumption of continuity has to be kept in mind despite that each grid point is storing the physical solution in discretized form. An interpolation method that expands the values of the discretized solution in the continuous solution, in a whole domain, models the concept of the continuum throughout the computational grid. Thus, the base for the continuum field analysis is formed and we can extract quantities related to different geometries as point, curve or surface fields.
1.3.1 Interpolation Method

The numerical treatment of the continuum model is based on interpolation methods, which are used in constructing new data points from the discrete set of known quantity fields' points. The continuum data function is approximated with the interpolation formula, which is defined with the set of discrete numerical values. Some numerical simulations compute the solution at the cell nodes and the scientific visualization algorithms use interpolation formulas to define the solution, for example to find a scalar or vector data, at any desired location inside the cell. In this section the interpolation formulas are defined for the coordinate transformations, Jacobian matrices and Metric tensors, see Figure 75, for the selected cell types, see section 1.1.1:

1D segment, \( U_1 \) in \( E^1, E^2, E^3 \)

2D triangle, quadrilateral, \( U_2 \) in \( E^2, E^3 \)

3D tetrahedron, pyramid, pentahedron, hexahedron, \( U_3 \) in \( E^3 \)

The interpolation is based on the parametric definition of the point location, including the mapping between the modeling and parametric space. Each cell has the following two important parameters:

- the dimension of the parametric space \( k \).
- the dimension of the modeling space \( n \)

The Euclidian space \( E^n \) with dimension \( n=1,2 \) or \( 3 \) is defined with variables \( x, y \) and \( z \) used as coordinates in the modeling space. The parametric space \( U_k \) with dimension \( k=1,2 \) or \( 3 \) is defined with variables \( u, v \) and \( w \).

![Parametric Space vs Modeling Space](image)

Figure 75: Coordinates transformations characteristics

The mapping from the parametric space \( U \) to the modeling space \( E \) is defined as:

\[
A_k^n : U_k \rightarrow E^n
\]

1.3.1 - 1

and in vector notation:

\[
x (x,y,z) = A [u (u,v,w)]
\]

1.3.1 - 2

There are several geometrical variables, see Figure 75, which has to be defined for the mapping \( A \), which represents the base for the coordinate transformation, and in addition, it is applied for the definition of the Jacobian matrix \( J \) and the Metric tensor \( G \).
It is assumed that the cell is simply connected, thus topologically, uniquely describing a single portion of the Euclidian space \([62]\). It is important to note that the topological structure of the cell, named cell topology is preserved by the mapping \(A\). The cell is a closure of an open connected set and it is assumed to be a closed including its boundary \(\partial U_k\). This means in particular that the cell nodes are included in the mapping. The cell boundaries in the parametric cell space \(\partial U_k\) are aligned with the constant coordinate axis or lines/planes with unit normal \(u(1,1,1)\). Before the mapping \(A\) is derived, a cell in modeling space must be numerically specified with all its constituting nodes. The cell boundary \(\partial U_k\) must be given in order to define the boundary mapping as:

\[ \partial A_k^u : \partial U_k \Rightarrow \partial E^n \]

1.3.1 - 3

before the extension to the cell interior is done. Each point in the parametric space \(U\) is mapped to a unique point in the modeling space \(E\). Thus, for every point \(u \in U_k\), there exist a unique point \(x \in E^n\) and for every point \(x \in E^n\), there exist a unique point \(u \in U_k\). Such mapping is smooth and non-singular within the cell and preserves in addition the cell orientation.

As it could be noted, the superscript and subscript indexing notation was used in this section to indicate the contravariant and covariant nature of respective coordinates transformations to keep it general \([77]\). From now on, we assume that the applied coordinate systems are orthogonal, which yields that the respective contravariant and covariant coordinates are equal, and we used only the subscript indexing.

**Isoparametric mapping and shape functions**

It is assumed that the isoparametric mapping \(A\) is a polynomial form defining a general scalar field \(f\) at the parametric coordinates \(u\), with a set of coefficients \(a\). The general formulations, which take into account the cell parametric dimension (exponent’s \(r, s\) and \(t\) are increased respectively for \(1D, 2D\) and \(3D\)) and the number of cell nodes (indexed by \(i\)) are:

1D: \(f = \sum_{i=0}^{\infty} a_i (u^i)^j = a_0 u^0 + a_1 u^1\)

2D: \(f = \sum_{i=0}^{\infty} \sum_{r+s \leq 2} a_i (u^r v^s)^j = a_0 u^0 v^0 + a_1 u^1 v^0 + a_2 u^0 v^1 + a_3 u^1 v^1\)

3D: \(f = \sum_{i=0}^{\infty} \sum_{r+s+t \leq 3} a_i (u^r v^s w^t)^j\)

For example, in the case of a quadrilateral cell, the 2D equation 1.3.1 - 4 can be satisfied for each cell node by forming a set of simultaneous equations, for which we assume that the function value \(f_N\) is known at each cell node, as follows:
\[
\begin{bmatrix}
    f_0 \\
    f_1 \\
    f_2 \\
    f_3 \\
\end{bmatrix} = 
\begin{bmatrix}
    1 & u_0 & v_0 & u_0 v_0 \\
    1 & u_1 & v_1 & u_1 v_1 \\
    1 & u_2 & v_2 & u_2 v_2 \\
    1 & u_3 & v_3 & u_3 v_3 \\
\end{bmatrix} \begin{bmatrix}
    a_0 \\
    a_1 \\
    a_2 \\
    a_3 \\
\end{bmatrix}
\]

or in the vector notation

\[
f_N = C a
\]

1.3.1 - 5

The unknown coefficients \( a \) can be calculated by finding the inverse of \( C \) when \( f_N \) is defined at cell node, as:

\[
a = C^{-1} f_N
\]

1.3.1 - 6

Once coefficient \( a \) is computed, it can be substituted in equation 1.3.1-4, and written as

\[
f = p a
\]

1.3.1 - 7

where \( p(u) = \{ 1, u, v, \ldots, u v, \ldots, u^2 \ldots \} \)

resulting in

\[
f(u) = p(u) C^{-1} f_N
\]

1.3.1 - 8

The shape function \( h \) is defined as

\[
h(u) = p(u) C^{-1}
\]

1.3.1 - 9

and when combined with the node solution \( f_N \) gives the polynomial form

\[
f(u) = \sum_{i=0}^{M-1} h_i(u) (f_N)_i
\]

1.3.1 - 10

where \( M \) indicates the number of cell nodes. This equation reveals important constraints of the shape function \( h \) expressed as:

\[
h_i = 1, \text{ for node } i \text{ and } h_j = 0, \text{ for } \forall \text{ node } j \neq i
\]

1.3.1 - 11

The shape function \( h \) has a value of unity at an arbitrary node \( i \) and is equal to zero at all other nodes. The variation of the solution along all boundaries is retained in order to satisfy condition for continuity. The outlined formulation has two known disadvantages [69]:

1. The inverse of \( C \) may not exist, and
2. The evaluation of \( C \) in general terms for all cell types involves considerable algebraic difficulties.

As the definition of the shape function \( h \) is necessary for all the elaborated cell types, it is appropriate to define \( C^{-1} \) more explicitly for the code implementation. This is accomplished by applying the interpolation polynomial in the Lagrange form [78, 79], which is well known numerical interpolation method for its systematic definition of shape functions. Lagrange’s polynomials \( L \) satisfy the constraint given in the equation 1.3.1-11 and the cell-to-cell continuity condition. For example, their definition is given in explicit form, in 1D coordinate space,
In the term $L^n_i$, $n$ stands for the number of subdivision in the cell along the parametric coordinate axis. As the analyzed cell types are linear the number of subdivision is 1 and it is constant for all of them. In the following equations such indices are removed and replaced with the node number associated with the parametric coordinate ($u,v$). The explicit form of the 2D shape function is defined in accordance with the analogy to 1D as follows:

$$h_i = h_w = L_u(u) L_v(v)$$

1.3.1 - 13

and 3D shape function as:

$$h_i = h_{uvw} = L_u L_v L_w$$

1.3.1 - 14

The following relations satisfy the shape function conditions 1.3.1-12 for different dimensions spaces

$$1D: h_i = 1 \quad 2D: h_i h_j = \delta_{ij} \quad 3D: h_i h_j h_k = \varepsilon_{ijk}$$

1.3.1 - 15

Lagrange polynomials provide characteristics to define non-linear cell types. As the development of scientific visualization system could be diversified into non-linear cells, the presented structure represents a good basis to capture future needs to develop interpolation methods, which will better approximate the imported results from numerical simulation in the visualization system.

As explained, the isoparametric mapping $A$ is based on simple products of Lagrange polynomials in the parametric cell space $U$ supported by values defined at cell nodes:

$$f = A(u) = \sum_{i=1}^{M} h_i(u) (f_N)_i$$

1.3.1 - 16

where $M$ is the number of cell nodes and $(f_N)_i$ are the nodes coordinates or solution. The mapping is called isoparametric because the coordinates and solution are treated with same the shape function. The simplest shape function of order 1 is defined for 1D as:

$$L_0(u) = \frac{u-u_1}{u_0-u_1} = \frac{u-1}{0-1} = 1-u$$

$$L_1(u) = \frac{u-u_0}{u_1-u} = \frac{u-0}{1-0} = u$$

1.3.1 - 17

Substituting 1.3.1-17 in equations 1.3.1-13, 1.3.1-14 and 1.3.1-15 the following result is obtained, respectively for 1D, 2D and 3D. The fully developed mapping for linear cell types follows:

**1D: segment**

$$A \equiv \sum_{i=0}^{1} h_i f_i^N \Rightarrow \sum_{i=0}^{1} a_i u^i = a_0 + a_1 u^1 = a_0 + a_1 u$$

$$A = h_0 f_0 + h_1 f_1 \quad \text{where} \quad h_0 \equiv L_0(u) = 1 - u$$

$$h_1 \equiv L_1(u) = u$$
and after simple algebraic manipulations, the \( a \) coefficients are defined as:

\[
\begin{align*}
    a_0 &= f_0 \\
    a_1 &= f_1 - f_0 \\
    A &= a_0 + a_1 u \\
\end{align*}
\]

This is the isoparametric mapping for segment cell. Follows the derivation of the isoparametric mapping for 2D cell types:

\[
A = \sum_{i=0}^{3} h_i(u) f_i = \sum_{i=0}^{3} a_i (u^v) \]

2D: quadrilateral

\[
\begin{align*}
    h_0 &= L_0(u) L_0(v) = (1 - u)(1 - v) \\
    h_1 &= L_1(u) L_0(v) = u(1 - v) \\
    h_2 &= L_0(u) L_1(v) = u v \\
    h_3 &= L_1(u) L_1(v) = (1 - u)v
\end{align*}
\]

and after simple algebraic manipulations the mapping coefficients are

\[
\begin{align*}
    a_0 &= f_0 \\
    a_1 &= f_1 - f_0 \\
    a_2 &= f_1 - f_0 \\
    a_3 &= f_0 f_1 + f_2 - f_3 = -a_1 + f_2 - f_3 \\
    A &= a_0 + a_1 u + a_2 v + a_3 u v
\end{align*}
\]

The isoparametric mapping \( A \) for triangles is a degenerated case of the quadrilateral one, where the shapes function of the dummy node \( h_D = h_{11} = 0 \) annihilate the influence of the \( a_3 \) coefficient. The coefficients for 2D triangle follow:

\[
\begin{align*}
    a_0 &= f_0 \\
    a_1 &= f_1 - f_0 \\
    a_2 &= f_2 - f_0 \\
    h_D &= h_{11} = 0 \quad \Rightarrow \quad u v = a_3 = 0 \\
    A &= a_0 + a_1 u + a_2 v
\end{align*}
\]

Note that the nodes indices are shifted according to the triangle node connectivity see Table 3: TRIANGLE skeleton table on page 32. The most complex isoparametric mapping in the context of this thesis is the 3D mapping of the hexahedron.
This mapping is fully developed before the specific mappings for tetrahedron, pyramid and pentahedron, cells as they are degenerated cases of the hexahedron one. The shape functions for each node are defined in Table 21, and the mapping \( A \) is:

\[
A = a_0 + a_1 u + a_2 v + a_3 w + a_4 u w + a_5 u v + a_6 v w + a_7 u v w
\]

<table>
<thead>
<tr>
<th>Shape function</th>
<th>1</th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>uv</th>
<th>uw</th>
<th>wv</th>
<th>uvw</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_0 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{000} )</td>
<td>(1 - u) (1 - v) (1 - w)</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>( h_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{100} )</td>
<td>u (1 - v) (1 - w)</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_2 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{110} )</td>
<td>u v (1 - w)</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_3 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{010} )</td>
<td>(1 - u) v (1 - w)</td>
<td>+</td>
<td>-</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>( h_4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{001} )</td>
<td>(1 - u) (1 - v) w</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_5 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{101} )</td>
<td>u (1 - v) w</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_6 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{111} )</td>
<td>u v w</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_7 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( h_{011} )</td>
<td>(u - 1) v w</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 21: Shape function for 3D isoparametric mapping

From the Table 21, the coefficients are calculated from the node values \( f \) and after simple algebraic manipulation they are defined as:

\[
a_0 = f_0 \\
a_1 = f_1 - f_0 \\
a_2 = f_2 - f_0 \\
a_3 = f_3 - f_0 \\
a_4 = f_0 - f_1 + f_2 - f_3 = -a_1 + f_2 - f_3 \\
a_5 = f_0 - f_1 - f_4 + f_5 = -a_1 - f_4 + f_5 \\
a_6 = f_0 - f_3 - f_4 + f_7 = -a_2 - f_4 + f_7 \\
a_7 = f_0 - f_1 - f_2 + f_3 + f_4 - f_5 - f_6 + f_7 = -a_4 + f_4 - f_5 + f_6 - f_3
\]

The coefficients are grouped to reduce the number of multiplication operations as:

\[
A = a_0 + a_1 u [a_4 + a_7 v + a_5 w] + [a_2 + a_6 w] + a_3 w
\]

1.3.1 - 21

The 3D cells are always embedded in the hexahedron, see figure above. As mentioned, degenerated cases of the hexahedron are tetrahedron, pyramid and pentahedron cell. The shape functions \( h_i \), containing the non existing nodes are removed from the isoparametric mapping \( A \) by making them equal to zero, equation 1.3.1 - 21. For the
tetrahedron the existing nodes are 0, 1, 3 and 4 of the hexahedron, thus the shape functions $h_2$, $h_5$, $h_6$, $h_7$ are zero, see Table 21:

\[
\begin{align*}
    &h_2=0 & h_5= u v - u v w & \text{from } h_6= u v w = 0 & \Rightarrow u v = 0 \\
    &h_3=0 & h_6= u v w & \text{from } h_5= u v w = 0 & \Rightarrow u v w = 0 \\
    &h_5=0 & & & \Rightarrow u v w = 0 \\
    &h_7=0 & h_7= v w - u v w & \text{from } h_6= u v w = 0 & \Rightarrow v w = 0
\end{align*}
\]

and the coefficients are:

\[
\begin{align*}
    a_0 &= f_0 \\
    a_1 &= f_1 - f_0 \\
    a_2 &= f_3 - f_0 \\
    a_3 &= f_4 - f_0
\end{align*}
\]

In the tetrahedron node notation the coefficients are:

\[
\begin{align*}
    a_0 &= f_0 \\
    a_1 &= f_1 - f_0 \\
    a_2 &= f_3 - f_0 \\
    a_3 &= f_4 - f_0
\end{align*}
\]

and the mapping is $A = a_0 + a_1 u + a_2 v + a_3 u v$

For a pyramid the complete base of the hexahedron is included and in this way the hexahedron node indexing is equal to pyramid ones. The last three shape functions $h_5$, $h_6$, $h_7$ are zero, see tetrahedron derivation, and the existing coefficients are:

\[
\begin{align*}
    a_0 &= f_0 \\
    a_1 &= f_1 - f_0 \\
    a_2 &= f_3 - f_0 \\
    a_3 &= f_4 - f_0 \\
    a_4 &= f_0 - f_1 + f_2 - f_3 = f_2 - f_3 - a_1
\end{align*}
\]

forming the mapping:

\[
A = a_0 + u (a_1 + v a_4) + v a_2 + w a_3
\]
The mapping is
\[ A = a_0 + u (a_1 + a_4 w) + v (a_2 + a_5 w) + a_3 w \]

The number of multiplication operations is reduced with adequate grouping of mapping coefficient, as shown in the Table 22.

<table>
<thead>
<tr>
<th>Operation</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell</td>
<td>Simple</td>
</tr>
<tr>
<td>T2N3</td>
<td>2</td>
</tr>
<tr>
<td>T2N4</td>
<td>3</td>
</tr>
<tr>
<td>T3N4</td>
<td>4</td>
</tr>
<tr>
<td>T3N5</td>
<td>5</td>
</tr>
<tr>
<td>T3N6</td>
<td>7</td>
</tr>
<tr>
<td>T3N8</td>
<td>12</td>
</tr>
</tbody>
</table>

*Table 22: Reduction of multiplication operations*

**Jacobian matrix**

The modeling space \( E \) is discretized with an arbitrary set of cells and it is global to all of them. The parametric space \( U \) is local to every cell. The isoparametric mapping \( A \), applied to a point \( P \), transforms its parametric coordinates \( u \) to modeling coordinates \( x \).

The cell coordinates are used as mapping support, equation 1.3.1 - 16, and the mapping \( A \) becomes the coordinate's transformation \( x(u) \), see Figure 76.

\[ x = x_i (u, v, w) \quad i = 1, 2, 3 \]

Such mapping hasn’t much value if the inverse mapping \( A^{-1} \), denoted \( u(x) \) doesn’t exist. The mapping \( u(x) \) allows backward coordinates transformation to the parametric space, as shown in Figure 76.

\[ u = u_i (x, y, z) \quad i = 1, 2, 3 \]
and can be found if \( x \) is single valued and continuously differentiable in the neighborhood of a point \( P \). Thus, provided that Jacobian matrix

\[
J = \frac{\partial x}{\partial u}
\]

has the Jacobian \( J \) which exists and doesn’t vanish

\[
J = \left| \frac{\partial x}{\partial u} \right| = \begin{vmatrix} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} & \frac{\partial x}{\partial w} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} & \frac{\partial y}{\partial w} \\ \frac{\partial z}{\partial u} & \frac{\partial z}{\partial v} & \frac{\partial z}{\partial w} \end{vmatrix}
\]

This implies the existence of the inverse \( J^{-1} \). \( J \) is calculated from \( A \), see equation 1.3.1 - 16, as

\[
J = \frac{\partial x}{\partial u} = \frac{\partial A(u)}{\partial u}
\]

For the cell origin \( J \) is defined with cell base vectors \([e_u, e_v, e_w]\), see Figure 76., as

\[
J = \begin{bmatrix} e_u & e_v & e_w \\ e_u & e_v & e_w \\ e_u & e_v & e_w \end{bmatrix}
\]

In \( E^3 \) the triad \([e_u, e_v, e_w]\) serves as a basis for \( U_3 \) provided that they are not coplanar.

\[
e_u \cdot (e_u \land e_w) \neq 0
\]

The three unit vectors have each, only one non vanishing component in \( U_3 \)

\[
e_u = e_{(1)} = (1,0,0)
\]

\[
e_v = e_{(2)} = (0,1,0)
\]

\[
e_w = e_{(3)} = (0,0,1)
\]

The suffixes to the \( e \) are enclosed in parenthesis to show they do not denote components. The \( j \)-th component of \( e_{ij} \) is denoted by \( e_{ij} \) and satisfies the following relation

\[
e_{ij} = \delta_{ij}
\]

In \( U_3 \) any vector \( a \) can be expressed in the form

\[
a = a_i e_{(i)}
\]

and the summation convention is also applied to suffixes enclosed in parentheses.

Isoparametric mapping for quantities is important, as it allows combined treatment of quantities interpolation and mapping between parametric and modeling coordinate’s spaces. When the modeling quantities have to be manipulated and therefore be defined in both coordinate spaces, the knowledge of \( J \) is required to define the quantity components in both spaces, while the quantity itself is invariant. These characteristics are used for vector line algorithm; see section 1.4.5, when the integration of the particle path is performed through the parametric vector field. Another application of the Jacobian occurs in the calculation of derived quantities, for
example vorticity, is another application of the Jacobian matrix. Follows the derivation of Jacobian matrices for predefined cell types:

Jacobian 1D matrix:

\[
J_{1D} = \frac{\partial A}{\partial u} \equiv \begin{bmatrix} \frac{\partial A}{\partial u} \end{bmatrix}
\]

1D segment \( A = a_0 + a_1 u \)

\[
J_{T1N2} = \begin{bmatrix} \frac{\partial A}{\partial u} = a_1 \end{bmatrix}
\]

Jacobian 2D matrix:

\[
J_{2D} = \frac{\partial A}{\partial u} \equiv \begin{bmatrix} \frac{\partial A}{\partial u}, \frac{\partial A}{\partial v} \end{bmatrix}
\]

2D triangle

\( A = a_0 + a_1 u + a_2 v \)

\[
J_{T2N3} = \begin{bmatrix} \frac{\partial A}{\partial u} = a_1, \frac{\partial A}{\partial v} = a_2 \end{bmatrix}
\]

2D quadrilateral

\( A = a_0 + a_1 u + a_2 v + a_3 u v \)

\[
J_{T2N4} = \begin{bmatrix} \frac{\partial A}{\partial u} = a_1 + a_3 v, \frac{\partial A}{\partial v} = a_2 + a_3 u \end{bmatrix}
\]

Jacobian 3D matrix:

\[
J_{3D} = \frac{\partial A}{\partial u} \equiv \begin{bmatrix} \frac{\partial A}{\partial u}, \frac{\partial A}{\partial v}, \frac{\partial A}{\partial w} \end{bmatrix}
\]

3D hexahedral

\( A = a_0 + u \{ a_1 + a_5 w + v (a_4 + a_7 w) \} + v (a_2 + a_6 w) + a_3 w \)

\[
J_{T3N8} = \begin{bmatrix} \frac{\partial A}{\partial u}, \frac{\partial A}{\partial v}, \frac{\partial A}{\partial w} \end{bmatrix}
\]

\[
\frac{\partial A}{\partial u} = a_1 + a_5 w + v (a_4 + a_7 w)
\]

\[
\frac{\partial A}{\partial v} = a_2 + u (a_4 + a_7 w) + a_6 w
\]
\[
\frac{\partial A}{\partial w} = a_1 + a_6 v + u (a_3 + v a_7)
\]

1.3.1 - 36

3D tetrahedron

\[
A = a_0 + a_1 u + a_2 v + a_3 w
\]

\[
J_{TNS} = \begin{bmatrix}
\frac{\partial A}{\partial u} = a_1, & \frac{\partial A}{\partial v} = a_2, & \frac{\partial A}{\partial w} = a_3
\end{bmatrix}
\]

1.3.1 - 37

3D pyramid

\[
A = a_0 + (a_1 + a_4 v) u + a_2 v + a_3 w
\]

\[
J_{TNS} = \begin{bmatrix}
\frac{\partial A}{\partial u} = a_1 + a_4 v, & \frac{\partial A}{\partial v} = a_2 + a_4 u, & \frac{\partial A}{\partial w} = a_3
\end{bmatrix}
\]

1.3.1 - 38

3D prism

\[
A = a_0 + u (a_1 + a_5 w) + v (a_2 + a_5 w) + a_3 w
\]

\[
J_{TNS} = \begin{bmatrix}
\frac{\partial A}{\partial u} = a_1 + a_5 w, & \frac{\partial A}{\partial v} = a_2 + a_5 w, & \frac{\partial A}{\partial w} = a_3 + a_4 u + a_5 v
\end{bmatrix}
\]

1.3.1 - 39

Metric tensor

The metric tensor is one of the basic objects in differential geometry [77, 80] and it is related to geometrical properties such as length, area or volume, respectively in 1D, 2D or 3D coordinate space. It is the ratio between the two coordinate systems for which the isoparametric mapping \(A\) and the Jacobian matrix \(J\) are defined. The metric tensor \(G\) can be written as

\[
G = J^T J
\]

1.3.1 - 40

This relation is important in the case when parametric and modeling spaces differ in dimension. For example, when a quadrilateral cell is placed in the 3D modeling space the Jacobian matrix is not of the square type:

\[
J = \begin{bmatrix}
\frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\
\frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \\
\frac{\partial z}{\partial u} & \frac{\partial z}{\partial v}
\end{bmatrix}
\]

1.3.1 - 41

This is a significant inconvenience for the calculation of the inverse Jacobian matrix \(J^T\) as in general \(J\) is a rectangular matrix. Let us consider its Moore-Penrose generalized inverse \(J^*_T\), see [81] for equalities on generalized inverses. The metric tensor \(G = J^*_T J\) is square and regular, thus its generalized inverse is equal to its ordinary inverse, so that
The inverse of the metric tensor \( G^{-1} \) can be explicitly calculated being by definition the square type matrix:

\[
G = \{g_{ij}\}
\]

where,

\[
g_{ij} = \sum_{k=1}^{n} \frac{\partial x_k}{\partial u_i} \frac{\partial x_k}{\partial u_j}
\]

or in the vector notation: \( x_{u_1} \equiv x_u \quad x_{u_2} \equiv x_v \)

\[
g_{ij} = x_{u_i} \cdot x_{u_j}
\]

The definition in vector notation shows that \( g_{ij} \) is the dot product of the tangent vector of the \( i \)-th coordinate axis with the tangent vector of the \( j \)-th coordinate axis analyzed from the modeling coordinates space.

The vector (cross) product of these two vectors \( x_u, x_v \) defines the normal \( n \) of the tangential plane at the cell node \( n_0 \).

\[
n = \frac{x_u \otimes x_v}{|x_u \otimes x_v|}
\]
The four nodes of the quadrilateral are defining the surface, see Figure 77. The surface point for which the normal exists is the regular surface point and has to satisfy the following condition:

\[ x_u \odot x_v \neq 0 \]  

If the above condition is satisfied, the vectors \( x_u \) and \( x_v \) are not collinear and they define the tangential plane. If the condition is not satisfied the surface point is singular and the Jacobian of such coordinates transformation is zero. Following the equation 1.3.1-41 the transpose Jacobian matrix is:

\[
J^T = \begin{bmatrix}
\frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} & \frac{\partial z}{\partial u} \\
\frac{\partial x}{\partial v} & \frac{\partial y}{\partial v} & \frac{\partial z}{\partial v} \\
\frac{\partial x}{\partial u} & \frac{\partial y}{\partial u} & \frac{\partial z}{\partial u}
\end{bmatrix}
\]

and applying it to the equation 1.3.1-40, we obtain the following expanded form:

\[
G = J^T J = \begin{bmatrix}
\left( \frac{\partial x}{\partial u} \right)^2 + \left( \frac{\partial y}{\partial u} \right)^2 + \left( \frac{\partial z}{\partial u} \right)^2 \\
\frac{\partial x}{\partial u} \frac{\partial y}{\partial v} + \frac{\partial y}{\partial u} \frac{\partial y}{\partial v} + \frac{\partial z}{\partial u} \frac{\partial z}{\partial v} \\
\frac{\partial x}{\partial u} \frac{\partial x}{\partial v} + \frac{\partial y}{\partial u} \frac{\partial y}{\partial v} + \frac{\partial z}{\partial u} \frac{\partial z}{\partial v}
\end{bmatrix}
\]

From which the \( G^{-1} \) can be calculated, and thus we have defined all the elements for calculating \( J^\dagger \) in the equation 1.3.1-42.

Some additional characteristics of the Jacobian matrix and Metric tensor determinants, defined as

\[ G = |G| \]

and, if the Jacobian matrix is of the square type, the metric \( G \) is defined explicitly with the Jacobian \( J \) as

\[ J = \sqrt{\det J}, \quad G = J^2 \]

For example, the calculation of cell length \( L \), area \( A \) and volume \( V \) is possible by knowing the Jacobian of isoparametric mapping:

\[
J = \frac{dV}{dV_0}, \quad J = \frac{dA}{dA_0}, \quad J = \frac{dL}{dL_0}.
\]

### 1.3.2 Derived Quantity

The derived quantities are the ones, which we are able to compute from the available data provided by an Input Model. As the inputted data are usually coordinates, scalar and vectors, by simple algebraic operations we can add, subtract, multiply and divide the existing quantities, defined as Unary Operations, for which a constants can be introduced, for example when scaling the coordinates. We can also consider Binary Field Operations when we use multiple quantities in an algebraic expression. The possibility to derive quantities from a vector field in terms of components and magnitude is also considered, and it is described in section 2.8, where a Symbolic Calculator is presented. In this section we will describe the isoparametric numerical model of the cell to be
applied when \textit{Gradient}, \textit{Divergence} and \textit{Curl} operators are computed for the related scalar or vector quantity field.

The scalar function $S$ of position $s$ when gradient operator $\text{grad} \ (\nabla)$ is applied to it produces a vector $\nabla S$.

\[
\mathbf{v} = \nabla S = \text{grad} \\
S = \frac{\partial}{\partial x} e_x + \frac{\partial}{\partial y} e_y + \frac{\partial}{\partial z} e_z
\]

1.3.2 - 1

There is also a need to calculate partial derivatives of the quantity, as divergence:

\[
\mathbf{s} = \nabla \mathbf{v} = \text{div} \ \mathbf{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}
\]

1.3.2 - 2

The important physical meaning for the divergence of the velocity field $\mathbf{v}$ is that it represents the relative rate of the space dilatation when computed along the particle trace. Consider the cell around the point $P$. By the fluid motion this cell is moved and distorted. As the fluid motion cannot break up by the continuity law its volume $dV_0 = J \ dV$ and hence:

\[
J = \frac{dV(t_0)}{dV(t)}
\]

1.3.2 - 3

defines the ratio of the cell volume at beginning and time $t$, called the dilatation or expansion [77].

Suppose a velocity vector field $\mathbf{v}(x)$ defined in a 3D Euclidian (modeling) space. The vorticity field $\mathbf{\Omega}(x)$ is the circulation of the velocity field $\mathbf{v}$ around a cell area taken perpendicularly to the direction of $\mathbf{\Omega}$, and $\mathbf{\Omega}$ is obtained by computing at each point the curl of the velocity field $\mathbf{v}$.

\[
\mathbf{\Omega} = \text{curl} \ \mathbf{v}
\]

\[
\Omega_x = \frac{\partial v_y}{\partial z} - \frac{\partial v_z}{\partial y} \\
\Omega_y = \frac{\partial v_z}{\partial x} - \frac{\partial v_x}{\partial z} \\
\Omega_z = \frac{\partial v_x}{\partial y} - \frac{\partial v_y}{\partial x}
\]

1.3.2 - 4

\textbf{Analytical Model}

If $\mathbf{v}$ is given as function in the parametric coordinate system $\mathbf{u}(u,v,w)$, with a well know link to modeling space $\mathbf{x}(x,y,z)$ through isoparametric mapping, see section 1.3.1 on page 74. The components of the velocity vector $\mathbf{v}$ can be written as:

\[
\mathbf{v}(\mathbf{u}) = [v_x(u,v,w), v_y(u,v,w), v_z(u,v,w)]
\]

1.3.2 - 5

and by the law of partial derivation, applied to Jacobian matrix, see section see section 1.3.1 on page 80:

\[
\frac{d}{dt}(J^{-1}) = \frac{d}{dt}\left(\frac{\partial u_i}{\partial x_j}\right) = \frac{\partial}{\partial x_j} \frac{du_i}{dt} = \frac{\partial v_i}{\partial x_j}, \quad i,j = 1,2,3
\]

1.3.2 - 6
and defining the partial derivative of the vector component along the modeling coordinate axis:

\[
\frac{\partial v_j}{\partial x_j} = \frac{\partial v_j}{\partial u_i} \frac{\partial u_i}{\partial x_j} + \frac{\partial v_j}{\partial u_2} \frac{\partial u_2}{\partial x_j} + \frac{\partial v_j}{\partial u_3} \frac{\partial u_3}{\partial x_j} = \frac{\partial v_j}{\partial u_k} \frac{\partial u_k}{\partial x_j}
\]

1.3.2 - 7

where \( u(u_1, u_2, u_3) \equiv u(u, v, w) \)

and for the specific case \( \frac{\partial v_y}{\partial z} \)

\[
\frac{\partial v_y}{\partial z} = \frac{\partial v_y}{\partial u} \frac{\partial u}{\partial z} + \frac{\partial v_y}{\partial v} \frac{\partial v}{\partial z} + \frac{\partial v_y}{\partial w} \frac{\partial w}{\partial z}
\]

1.3.2 - 8

or putting it into the form of matrix multiplication

\[
\frac{\partial v}{\partial x} = \frac{\partial v}{\partial u} J^{-1}
\]

1.3.2 - 9

and expand the notation

\[
\left[ \begin{array}{ccc}
\frac{\partial v_x}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_z}{\partial z} \\
\frac{\partial v_x}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_z}{\partial z} \\
\frac{\partial v_x}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_z}{\partial z}
\end{array} \right] \left[ \begin{array}{ccc}
\frac{\partial v_x}{\partial u} & \frac{\partial v_y}{\partial v} & \frac{\partial v_z}{\partial w} \\
\frac{\partial v_x}{\partial u} & \frac{\partial v_y}{\partial v} & \frac{\partial v_z}{\partial w} \\
\frac{\partial v_x}{\partial u} & \frac{\partial v_y}{\partial v} & \frac{\partial v_z}{\partial w} \\
\frac{\partial v_x}{\partial u} & \frac{\partial v_y}{\partial v} & \frac{\partial v_z}{\partial w}
\end{array} \right] \left[ \begin{array}{ccc}
\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\
\frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\
\frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z}
\end{array} \right]
\]

The components of the \( \frac{\partial v}{\partial x} \) matrix are applied in the equations for gradient divergence and vorticity. For the equation 1.3.2 - 8 the remaining part \( \frac{\partial v}{\partial u} \) is calculated locally for each cell node when the inverse of Jacobian \( J \) exists.

**Numerical Model**

The velocity field is given at cell nodes and for each node \( \frac{\partial v}{\partial u} = \frac{\Delta v}{\Delta u} \) components are calculated for all cell types.

**1D segment**

\[
(\frac{\Delta v}{\Delta u})_{j,0} = v_j - v_0
\]

**2D triangle**

\[
(\frac{\Delta v}{\Delta u})_{j} = (v_i)_j - (v_j)_0 \quad (\frac{\Delta v}{\Delta u})_{k} = (v_i)_k - (v_j)_k
\]
2D triangle

\[ \frac{\Delta v_u}{\Delta u} = (v_u)_2 - (v_u)_0 \]
\[ \frac{\Delta v_v}{\Delta u} = (v_v)_2 - (v_v)_0 \]
\[ \frac{\Delta v_u}{\Delta v} = (v_u)_2 - (v_u)_0 \]
\[ \frac{\Delta v_v}{\Delta v} = (v_v)_2 - (v_v)_0 \]

<table>
<thead>
<tr>
<th>k</th>
<th>j</th>
<th>$\frac{\Delta v_i}{\Delta u}$</th>
<th>$\frac{\Delta v_i}{\Delta v}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1-0</td>
<td>2-0</td>
<td></td>
</tr>
</tbody>
</table>

2D quadrilateral

\[ \left( \frac{\partial v_i}{\partial u} \right)_{j,0} = (v_i)_j - (v_i)_k \]
\[ \left( \frac{\partial v_i}{\partial u} \right)_{2,0} = (v_i)_j - (v_i)_k \]
\[ \left( \frac{\partial v_i}{\partial u} \right)_{3,0} = (v_i)_j - (v_i)_k \]
\[ \left( \frac{\partial v_i}{\partial v} \right)_{2,0} = (v_i)_j - (v_i)_k \]

<table>
<thead>
<tr>
<th>k</th>
<th>j</th>
<th>$\frac{\Delta v_i}{\Delta u}$</th>
<th>$\frac{\Delta v_i}{\Delta v}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-0</td>
<td>3-0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1-0</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2-3</td>
<td>2-0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2-3</td>
<td>3-0</td>
<td></td>
</tr>
</tbody>
</table>

3D tetrahedron

\[ \left( \frac{\partial v_i}{\partial u} \right)_{j,0} = (v_i)_j - (v_i)_k \]
\[ \left( \frac{\partial v_i}{\partial u} \right)_{2,3} = (v_i)_j - (v_i)_k \]
\[ \left( \frac{\partial v_i}{\partial u} \right)_{2,0} = (v_i)_j - (v_i)_k \]
\[ \left( \frac{\partial v_i}{\partial w} \right)_{2,3} = (v_i)_j - (v_i)_k \]

<table>
<thead>
<tr>
<th>k</th>
<th>j</th>
<th>$\frac{\Delta v_i}{\Delta u}$</th>
<th>$\frac{\Delta v_i}{\Delta v}$</th>
<th>$\frac{\Delta v_i}{\Delta w}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-0</td>
<td>2-0</td>
<td>3-0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1-0</td>
<td>2-0</td>
<td>3-0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1-0</td>
<td>2-0</td>
<td>3-0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1-0</td>
<td>2-0</td>
<td>3-0</td>
<td></td>
</tr>
</tbody>
</table>
The linear derivatives for the considered cell types are introduced in the node connectivity algorithm for the calculation of the uniquely defined quantity at each cell node.
1.4 Extraction Algorithms

The numerical data model with geometry and quantity definitions requires effective ‘extraction algorithms’ for filtering and manipulating large data sets. Extracted data sets, to be visualized, are modeled as representations of surface, curve or point type. The numerical probes are represented by points, lines or planes; they are used to intersect grid geometry and quantity fields. In practice, the most general 4D data set is expressed as:

\[ q = q(x, y, z, t) \]

and its form in nD space is extended and given by:

\[ q = q(x_1, x_2... x_n) \]

The quantity \( q \) can be associated with a scalar, vector or tensor field. Visualization techniques in order to make extracted data set displayable require geometrical representations in 2D or 3D space. The mapping consists in converting the data set into a geometrical entity, like a surface, and to associate color to quantity value in order to provide ‘added-value’ to the visualization. The possibilities of extraction algorithms are multiple since they can create:

- surfaces: cutting planes or iso-surfaces,
- curves: iso-lines, sections and particle traces,
- points: locations of local values.

In extraction algorithms, a threshold, which is the difference between the requested and the node value, is applied to each mesh node. The cells, where at least one of the nodes thresholds has a different sign than the others cells’ nodes are identified as ‘intersected cells’, to which the cell intersection pattern is going to be applied. The cell connectivity model is essential for the extraction algorithms, because it allows the application of the marching cell algorithm for cutting plane and iso-surface algorithms. The same observation can be made when comparing the section and the iso-lines algorithms. The only difference is the lowering of the parametric dimension of the zone from which the extraction is made.

The calculated threshold can be the difference between the given values in respect to geometry or quantity values at the cell nodes. The threshold property defines:

- geometry for cutting plane, section and local value
- quantity for isosurface, isolines and particle traces.

This chapter makes difference between geometry/quantity extraction algorithms. It defines two types of marching cell algorithms based on the above mentioned threshold property. In addition follows the section on threshold regions where the sub zones are extracted from zones. The point location and particle trace algorithm for volume and surface are closing this chapter. They are treated separately because they require the point cell relationship with the objective to state that the point is in or out of the cell.

The following sections describes the marching cell algorithm which represents a common part of the cutting plane and isosurface algorithms, which are the most important interactive visualization tool to investigate volumetric data.

1.4.1 Marching Cell Algorithm

The marching cell algorithm is the core part of all extraction algorithms and represents the generalization of the marching cube algorithm [82]; it performs the following five tasks:

1. Identification of seed cells, which are initial cells from which the cell traversal starts,
2. Identification of intersected cells,
3. Marching through cells applying connectivity,
4. Application of intersection pattern with creation of new cells,
5. Creation of the appropriate zone from the collection of created cells.

The identification of intersected cells is based on the “point-to-plane distance” for the cutting plane and section, where as for the iso-surface and the iso-line it is the difference between the node value and the value provided by the user input. The cutting plane threshold explanation follows in more details as the cutting plane input has more parameters than iso-surface one, due to the interactive translation and rotation functionality. Each interactive re-positioning of a cutting plane requires the recalculation of geometry and quantity data. To identify an intersected cell, its nodes are labeled according to the distance sign of the nodes with respect to the cutting plane. The equation of the cutting-plane $S$ is:

$$S(x) = (x - x_0) \cdot n = 0,$$

where $x(x_0, y_0, z_0)$ is an arbitrary point of the plane and $n(n_x, n_y, n_z)$ is the unit-vector normal to the plane. A point $x$ for which $S(x) > 0$ is conventionally at a ‘positive distance’ from the plane. The plane equation is written in (right-handed) Cartesian coordinates in the form:

$$A x + B y + C z + D = 0 \quad \text{or} \quad A x + D = 0 \quad 1.4.1 - 1$$

where the vector $A (A,B,C)$ is along the normal to the plane. The distance $d_0$ between the point $x(x_0, y_0, z_0)$ and the plane:

$$d_0 = \frac{A x_0 + B y_0 + C z_0 + D}{\sqrt{A^2 + B^2 + C^2}} = \frac{A r_0 + D}{|A|} \quad 1.4.1 - 2$$

where the sign of the square root is chosen to be opposite to that of $D$. The definition of the distance sign is:

$$\text{sign} (d_0) = \text{sign} (A x_0 + D) \quad 1.4.1 - 3$$

for all cell nodes. $\text{sign} (d_0)$ defines on which side of the plane the node $r_0$ is located.

Each node position vector is projected on the normal $n$ of the cutting plane in order to determine its $\text{sign} (d_0)$ (positive or negative). The set of the cell nodes $\text{sign} (d_0)$ describes the cell intersection pattern, as shown in the lookup table where all the possible intersection patterns are labeled and defined, see Section 1.2.6.

$$0 \geq \text{sign} \equiv \text{false} \quad 0 < \text{sign} \equiv \text{true}$$

![Figure 78: Possible singular cases of intersections with a node, an edge and a face](image)
Cells which have nodes with positive and negative distance signs are intersected by the cutting-plane (cells where distance signs are all positive or all negative are not intersected).

The cutting-plane algorithm is improved by traversing the domain boundary, in order to avoid scanning all the domain cells, as shown in Figure 79. Every domain has one or more boundaries, which in 3D space consists of one or more surface boundaries. The finding of seed cells is specific to the extraction algorithms. The zone representation is based on B-rep model where each zone has its own boundary. From the seed cells the marching cell algorithm starts. We apply the extraction algorithm to the boundary, which is obviously faster than traversing the whole zone. We identify the intersected boundary cells. This is done in 2D, where surfaces are bounded by curves or in 3D where volumes are bounded by closed surfaces (see Figure 79). If we intersect a 3D surface, it is not sufficient to intersect its boundaries, as it can happen that the plane intersects the surface but not its curve boundary. However, the existence of the intersection with a surface boundary implies the existence of surface/plane intersection. It is obvious that a curve/line intersection in 2D (or a plane/surface intersection in 3D) detects all the points (or curve cells) necessary to complete the marching-cell algorithm.

A difference between the cutting-plane/section and the iso-surface/iso-line algorithms is that the former includes the generation of seeds from its boundaries. For the volume boundaries, the section algorithm is applied to find the seed cells. A main advantage is the improved performance of the cutting-plane algorithm compared to the iso-surface one, since the number of cells and nodes to be analyzed is one order of magnitude smaller. Surface cells have 3 or 4 nodes (triangle or quadrilateral), where volume cells have 4 or 8 nodes (tetrahedral or hexahedral). The number of cells that are visited on a 3D structured grid scales like $O(N^3)$ for the iso-surface algorithm and like $O(N^2)$ for the cutting-plane algorithm (where $N$ is the number of traversed cells).

The surface extraction procedure can be performed by the cutting-plane and the iso-surface algorithms. Both algorithms produce ‘unstructured surfaces’, i.e. surfaces composed of the cells constructed by the cell-intersection pattern. To reduce the time it takes to identify the intersected cells, a technique called the marching-cell algorithm is used. The marching cell algorithm exploits the fact that a plane, intersecting a mesh (for example a structured mesh of cubic cells) cuts only a limited number of cubes (Figure 80). Inside the domain, it is possible to start from any intersected cell and to ‘march’ to the next intersected cell until all intersected cells are visited. Connectivity is exploited to avoid scanning all the cells of the mesh. The marching-cell algorithm improves the sequential algorithm, which is not the case for the parallel one [43].

The cutting-plane algorithm comprises the following steps:

- find all intersected cell on the volume boundary and create seeds set
- while(seeds not empty)
  - create node mask

Figure 79: Seed cells from boundary
• for each intersected cell:
  • compute intersection polygon applying cell intersection pattern.
  • add connected cells to the seeds if not yet intersected.

The algorithm selects the appropriate cell type based on the parametric cell dimension and the number of cell nodes. The node mask defines the cell intersection pattern and the intersection can be calculated. To keep track in the marching-cell algorithm the set of intersected cells $seeds$ must be removed from the seed set of cells to avoid possible duplication in processing of intersected cells. For each intersected cell, the edges intersections with the cutting plane are computed. The nodes on the intersected edges define polygons, see Section 1.2.6, which all together define the cutting plane. In principle, these polygons are used for defining the graphical output primitive, see Section 3.7.2. The intersection of edge and plane is calculated in parametric form using a linear interpolation between two edge nodes; the resulting ratio is applied to compute the quantity value at the point of intersection. When the surface is kept for further processing, the created parametric field (nodes and the quantity parameter) is saved in order to speed up the recalculation of other available quantities.

The orientation of all created cells must be the consistent, because of the graphics rendering techniques, which require the normals definition as input to the shading and contouring algorithms. The orientation is implicitly given in the lookup table, where the order of triangle and quadrilateral nodes defines the local coordinate system and thus the normal to the generated cells (see Section 1.2.6). The same approach is used for pyramids, prisms and hexahedrons. Situations with intersected patterns with several internal pieces may be more complex and result in larger lookup tables, but processing remains extremely rapid with no need for conditional branching code.

This algorithm has 2 modes of operations: display and save. It is not always necessary to define the topological relationships and save the surface topology. In the display mode, the sole issue is to display the colored contour map or vector field. If other representations are needed -- like distribution of a scalar quantity in a $x$-$y$ plot -- the cutting plane must be saved, which implies the creation of an unstructured surface (made up of triangles and
quadrilaterals). The display mode manipulates polygons only so as to speed up interactive processing. The saving process calculates the needed cell-connectivity information, surface boundary and quantity parameters taking into account the original 3D domain from which the surface is extracted. The parametric field allows recalculating the surface field for an arbitrarily chosen quantity, thus reducing the processing and memory requirements to perform such operation. The save mode algorithm is decomposed in the following steps:

- find all intersected cells on the boundary and create seeds set
- while(seeds not empty)
  - initialize set of cells to be intersected \( xcells \) and add one seed cell to it
  - initialize collection of intersected cells
  - while(\( xcells \) not empty)
    - create node mask
    - for each intersected cell:
      - compute intersection polygon applying cell intersection pattern.
      - partition the polygon and compute connectivity
      - add polygon partition to the collection of intersected cells
      - add connected cells to the \( xcells \) if not yet intersected and remove from seeds if exist
    - create topology from the collection of intersected cells

After calculating edge intersections and polygon partitions, cell connectivity is examined. At the time one computes the intersections in a given cell; its connectivity is not necessarily locally definable since it can be that the adjacent cells are not yet constructed. Overall connectivity is preserved through ‘face indexing’, an indexing convention that depicts (internal and external) cell connectivity. Face indexing consists of indexing the virtual faces between polygonal cells with integers which are bigger than the number of faces of the cell. Each additional cell in the polygon is given an incremental index which is bigger than the number of cells in the zone. The resulting collection of all intersected cells must be further processed by topology algorithms that establish the topological description of the cutting plane as an unstructured surface. The connectivity of the intersected cells is defined applying the global SupZone indexing, to which internal indexing is added. The collection of cells is sorted using the global indexing. This characteristic allows a fast transformation where for each cell with greater index than the current one the cell connectivity is calculated with preserving local indexing.

The node-reduction algorithm takes into account the relation between node and cell, where each node can build more than one cell (1.2.4 Node Topology). The alternative is to eliminate the multiple intersections of the same edge. The detection of an intersected edge and the addition of new one in an ordered set imposes local node indexing (see Section 1.2.3 Cell Connectivity). The intersected supZone edges are stored in the set based on edge-nodes key. If an edge is intersected, the result of intersection is reused and the unique node indexing is preserved. For a convex domain, there is one single connected set of cells, defining the arbitrary cutting-plane surface. If the domain is not convex, multiple cutting-plane regions may exist, depending of the position and orientation of the cutting plane (see Figure 81).

![Figure 81: Cutting plane example of multiple connected topology](image-url)
The iso-surface algorithm computes the unstructured surface for a given constant scalar value. An iso-surface is not necessarily a plane and can have more general shape. The iso-surface algorithm differs from the cutting-plane in two respects: first, all the cells are checked; second, the difference between the scalar and threshold values replaces the distance-sign value. The algorithm is decomposed in following steps:

for each node find threshold
for each cell
    create node mask
    for each intersected cell:
        compute intersection polygon applying cell intersection pattern.

The save mode algorithm is decomposed in the following steps:

for each node find threshold
for each cell
    find intersected cell from not inspected cells
    initialize xcells collection with intersected cell
    while(xcells not empty)
        create node mask
        for each intersected cell:
            compute intersection polygon applying cell intersection pattern.
            partition the polygon and compute connectivity
            add polygon partition to the collection of intersected cells
            add connected cells to the xcells if not yet intersected and remove from seeds if exist
        create simply connected region from the collection of intersected cells

The 3D surface generated by this algorithm is ‘unstructured’; it is constructed in exactly the same manner as in the case of the cutting-plane. The marching-cell algorithm cannot be readily applied if the iso-surface has disconnected regions: as shown in Figure 82. The disconnected components may be open- or closed-type surfaces; indeed we can group any number of disconnected iso-surface parts into a set, which the user still refers to as a single surface. An iso-surface which has been saved has the functionality to ‘recreate’ any other quantities at its nodes; it is classified as a ‘parametric field’. The nodes of the original domain determine the intersection edges, together with the linear interpolation coefficient.

Distinguishing between structured and unstructured meshes is important when it comes to considering memory requirements. Unstructured meshes require more storage than structured ones, since one needs cell-connectivity to be defined explicitly. There is no significant difference between structured and unstructured meshes from the algorithmic point of view since intersection calculations are based on the cells lookup tables in both cases. The output of the cutting-plane algorithm is necessarily an unstructured surface, whether the base mesh is structured or not. An algorithmic difference exists, however, in how connectivity information is accessed. For structured
meshes, cell-connectivity is explicit to and deduced from the cell ordering sequence; this is in contrast to the case of unstructured meshes where cell-connectivity is provided as input data.

### 1.4.2 Threshold Regions

The threshold algorithm is an efficient extraction technique used in visualization; it can be applied in the parametric (indexing), modeling or quantity spaces. The idea is to create a SubZone (SubField) from the original Zone (Field).

The threshold in the index space is commonly used for specifying surface parts on the structured surfaces, when we define boundary conditions, for example a solid boundary. In that case, the imposed index threshold is the span between the two $I$ and $J$ indices. Since the indices are integer numbers, the resulting extraction is the structured 2D surface. A similar logic is applied for extracting edges and nodes. The difficult part is to re-create the subsurface because the nodes and the edges can determine distinct surface topologies. Such index based threshold can be applied to volume, surface or curve data sets (just not on point sets since one cannot construct SubPoint as SubZone). SubSurfaces and SubCurves are examples of SubZones for (un)structured grids (see Section 1.2.2).

In the geometrical space, one handles ranges of coordinates. A thresholded region can include part of a cell. The new SubCells must be completely defined as ‘polyline’, polygon or polyhedron; the goal is to be able to handle these poly-entities with the available graphic primitives.

The Quantity Threshold representation is a frequently applied representation which extracts SubSurfaces between 2 limiting iso-lines. It permits considerable reduction of the generated graphics (compared to the complete surface color-mapping) and enables the analysis of multiple surfaces. The threshold algorithm removes the possible drawback of having a surface masking another one. The threshold algorithm for both scalar and vectors quantities requires:

- the variable threshold type,
- identification of cells containing the threshold,
- construction of the SubZone from extracted cells

![Figure 83: Ten different thresholds in a triangle](image-url)
The generation of the SubCells is determined by the ‘cell threshold pattern’. The cell threshold pattern depends on the node threshold label -- which can be in any one of the 3 states: ‘included’, ‘excluded min’ or ‘excluded max’. For triangular and quadrilateral cells, the threshold analysis proceeds in much the same way as the cell intersection pattern defined with the binary node mask; here the processing is ‘n-ary’ instead of binary, and following equation applies:

\[
label \ index = s_0 3^0 + s_1 3^1 + \ldots + s_i 3^i + \ldots + s_{n-1} 3^{n-1}
\]

The number of possible threshold patterns for a cell with \(n\) nodes is given by:

\[
number \ of \ threshold \ patterns = 3^n
\]

so that for a triangle there are 27 threshold possibilities, and 81 for a quadrilateral cell; some of them are shown in Figure 83 and Figure 84. By imposing that the internal area of a threshold pattern is always a single connected region, one obtains patterns like those illustrated in Figure 85.
1.4.3 Curve Extraction: section and isoline algorithm

The logic of the cutting-plane and the iso-surface algorithms is readily applied to the section algorithm which simply lowers the parametric dimension of an intersected zone from volume to surface. An optimization is possible for 2D projects since a 2D domain is always an open surface and its boundaries are always intersected. The connectivity of the extracted zone is solved by reusing the connectivity of the involved zone cells.

A unified approach for the cutting-plane and the section algorithms has been constructed [75]. The section algorithm determines the surface/plane intersection and provides the quantity distribution in the created section; the section algorithm is the mere 2D version of the cutting-plane algorithm.

Isolines

The isoline representation is the primary technique for displaying scalar field information. The isoline algorithm calculates the curves that connect the surface points where the value of the scalar quantity is equal to a given input value; in effect, it is the 2D version of the iso-surface algorithm.

The scalar field, known by its values at the nodes of the cell, is filtered against the calculated intersection mask so as to detect a sign change; if this occurs, the cell is known to contain the isoline. The intersection is calculated by applying a linear interpolation between the nodes of the edge:

\[ \frac{q_{iso} - q_1}{q_2 - q_1} = \frac{x_{iso} - x_1}{x_2 - x_1} \]

When a cell is investigated, all its nodes are tested against the criterion \( q_n \cdot q_{iso} > 0 \), and the calculated sign is applied in the definition of the cell intersection mask. The intersection mask is used to find the intersection pattern in a cell lookup table. For surface cells, 2 cell tables are defined, the first one for triangles and the other one for quadrilaterals. There can be 0, 2 or 4 intersections for a quadrilateral. The ‘ambiguity case’ is when 4 intersections are detected; in such case, the 4 possibilities are drawn (see Figure 85).

There is no ambiguity case for the triangle. The marching algorithm generates open- and closed-curve types. Topologically, open-curves are always connected to the surface boundary, whilst closed-curves can always be transformed to a circle. A linear interpolation between edge node values is also used for defining the isoline colors (in relation to the color map).

There are several ways of applying the isoline algorithm. The straightforward method consists of traversing all surface cells for which the resulting isoline is created when connecting the intersected cells. When displayed, the resulting line segments are perceived as a map of open- and closed-type isolines. In this approach, the isoline connectivity table is not created which simplifies the algorithm but increases the calculation overhead, since every isoline point is shared by two adjacent cells and needs to be calculated twice. Hence the isolines are drawn with duplicated set of points, and this increases the display time, which can be problematic when interactive viewing actions like rotation and zooming are performed.

![Node traversal in the isoline algorithm for unstructured and structured surfaces](image-url)
The threshold sign is checked at each cell node. The node calculations are performed repeatedly since one node can be included in several cells (see Figure 86). On a structured surface, a node internal in the surface is shared by four adjacent cells; on an unstructured surface, a node may be shared by more than 4 cells.

There are two types of isolines that are surface curves (see Figure 87):
1. open isolines whose starting and ending points are on the surface boundary,
2. closed isolines that are parts of the surface interior.

The isoline algorithm involves edge detection:
- for each cell
- for each edge
- for each node

To generate curves for which there are no duplicated or disconnected points, the following algorithm is applied:

1. generation of a collection which contains at least one point on each closed isoline;
2. marching trough the surface grid starting from the boundary (case of open isolines);
3. marching trough the surface grid starting from the points in the list of remained isoline points, applied to define closed isolines.

Starting from the intersected boundary cells, one ‘follows’ the isoline by the existing cell-edge combination in the internal #cell-edge list, from which the combination is removed, when the isoline point becomes part of the isoline curve. The marching algorithm continues until the boundary-cell edge is reached. Each isoline point is added to the isoline curve. The next point defining the isoline is found from the cell intersection pattern following the cells connectivity. The ambiguity cases (if any) are handled without special attention. The ambiguity is locally checked if such cell pattern is met. When all the located boundary cells are removed from the list, all open isolines of the traversed surface are found.

The remaining internal cell edges are used in the same way as the boundary ones that have been intersected during the first cell-edge test. If that combination is found the isoline is closed. When the collection of internal cell-edges is empty, it means that all the closed isolines have been found.
1.4.4 Point Extraction: local value algorithm

Point extraction is computed by certain visualization tools, which require the identification of a surface cell containing the point indicated by the user. These visualization tools are: local value, local vector, local isoline, two points of the section, starting point of the particle trace and picking of the surface.

The point extraction algorithm solves the well-known geometrical problem called the point location problem. [83, 84] The algorithm applies transparently to 2D- and 3D-surfaces. The 2D problem can be simplified to the problem of 2 lines intersecting; the extension to 3D requires calculating the line at the intersection of two planes (see Figure 88). In all cases, the surface is intersected with the objective to find the intersected surface cell in which the point is located and the point’s coordinates on the surface itself. For an open surface, the algorithm can be optimized to avoid the full traversal of all the surface cells by finding the boundary cells which are intersected. This is the same idea as the one used for the section algorithm (see Section 1.4.3). The boundary cells that are found are used as the starting points for the marching-cell algorithm which then proceeds towards the interior of the surface. Since a surface consists of connected cells, cell connectivity can be used to identify the adjacent cells and the intersected edge uniquely identified cell to start the marching cell algorithm. The intersected cells are then ones more intersected with a second, auxiliary plane (see Figure 89). If there is no intersection with the auxiliary plane, the inputted point is not located in the surface cell and the algorithm continues to search in the direction prescribed by the section plane. This algorithm is an extension of the plane/surface intersection algorithm, with the added ‘point inclusion test’ which is performed for each cell intersecting with the auxiliary plane.

Figure 88: Line-Surface intersection concepts

Figure 89: Line Surface multiple intersections and possible traversals
The algorithm is decomposed in an ‘outer part’, which controls the traversal of the cells and an ‘inner part’ which deals with line/plane intersections and the point-inclusion test. The line/plane intersection algorithm is applied for:

- the boundary curve when curve cells are tested against possible section plane intersection
- and during the surface cells traversal where the cell edges are intersected.

The intersection point at the cell boundary (set of edges) is identified through the set of constraints applied to the point inclusion test. This part of the algorithm includes a combination of boundary constraints checks and line/plane intersections with surrounding cell edges. If the point is found to be in the cell, it is added to the set of intersections points, and traversal is pursued till all the seed cells are visited.

Inside the located cell, a Newton-Raphson method is applied to define more precisely the surface (see page 111). The point-location algorithm is decomposed in:

- line-plane intersections,
- point inclusion test,
- Newton Raphson method.

**Line Plane Intersection**

As mentioned line-plane intersection algorithm has several applications in the identification of region internal to the specific discretized zone. The examples are the line-surface intersection, and clipping algorithms.

When the particle trace integration generates the point outside the cell, the intersection of the trace with the cell boundary is found, and the integration continues in the adjacent cell. The last two points of the particle trace represent the oriented line segments, called ray. The ending point of the ray exerts the point inclusion test, which defines if the point is inside or outside of the cell. The side effect of the testing is the intersection point with the cell boundary, where a ray exits the cell.

In the clipping algorithms the sections or surfaces are imposed by predefined limits. A parametric line clipping algorithm, [6, 7, 85], is extended to different cell types in 2D and 3D parametric cell space, (intersection with a cell edge or face). For 2D the algorithm is simplified with two 2D lines intersection between the cell edge and the ray shown in Figure 90.

![Figure 90: Cell ray intersection](image)

The two points of the oriented segment \( p_0 \) and \( p_1 \) define the ray which intersects the cell boundary. The line or the plane \( P \) is intersected with ray \( L \), respectively in 2D and 3D cell space.

The ray \( L \) is defined parametrically as
The plane is defined with normal \( n \) and the point \( r_p \) lying inside the plane \( P \). The plane equation in the point-normal form is:

\[
P \equiv (r - r_p) \cdot n = 0
\]

In Figure 91, the imaginary line/plane \( P \) divides the cell plane into the outside and the inside half plane. The outside is indicated with positive direction of the normal \( n \). If the point \( p \) of the ray \( L \) is introduced in the equation 1.4.4 - 2 we define the half-space in which the line point is located:

\[
\begin{align*}
n \cdot \left( r_p - p \right) &< 0 & \text{outside half plane} \\
n \cdot \left( r_p - p \right) &> 0 & \text{inside half plane}
\end{align*}
\]

When the equation 1.4.4 - 1 is introduced in equation 1.4.4 - 2 the parameter \( t \) for the edge/face intersection point \( p_x \) is:

\[
t = \frac{n \cdot (r_p - p_o)}{n \cdot (p_1 - p_0)} \\
\text{where } t > 0 \text{ and } dp \neq 0.
\]

The valid value of \( t \) can be computed if the denominator is non zero. The algorithm checks that:

- \( n \neq 0 \), \( dp \neq 0 \), thus \( p_1 \neq p_0 \)
- \( n \neq 0 \), thus the edge and the segment are not parallel

\[
\begin{align*}
n \cdot (r_p - p_x) &< 0 & \text{outside half plane} \\
n \cdot (r_p - p_x) &> 0 & \text{inside half plane}
\end{align*}
\]
and the intersection point is at:

\[ p_s = p_0 + \frac{n(r_p - p_0)}{n \cdot dp} \cdot dp \]

1.4.4 - 5

Point location algorithm

The point location algorithm defines if the ending point of the ray is inside or outside the cell. In addition, if the point is outside the cell, it defines the intersection point of a ray with the cell boundary. The cell boundary is edge for 2D zones and face for 3D zones. The boundary is applied for the identification of the neighboring cell, see Figure 93. The intersection point can be transformed into the neighboring cell parametric coordinates, without the need to apply the time consuming Newton-Raphson algorithm, as discussed in the next section. The objective is to reduce the number of tests necessary to state if the point is in or out for the analyzed cell.

![Figure 93: Rays and intersections with extended cell boundary](image)

The point inclusion algorithm is based on the ray which is checked against several constrains defined for each cell type. The ray is defined with the starting point \( p_0 \) which is always inside the cell and the ending point \( p_1 \) for which the point inclusion algorithm is performed. The boundaries are extended in order to apply the line-plane/line-line intersection algorithm. The parameter \( t \), equation 1.4.4 - 4, is calculated for each boundary and the solution is the intersection point with minimum \( t \).

![Figure 94: Boundary normals for different parametric dimensions](image)

The extended cell boundaries split the external cell neighborhood in several regions. In order to define the ray exit point the extended boundaries define constrains which defines these regions. The number of regions \( R \) is:

\[ R = 2^C, \quad \text{where } C \text{ is the number of constraints.} \]

1.4.4 - 6
The ray ending point is checked against cell’s boundary which is further refined with the intersection of the ray with the boundaries of the cells. As an edge is a part of the boundary of the 2D cell and the face is part of the boundary of the 3D cell the plane line intersection algorithm, see page 101, is applied to find the point where the ray leaves the cell, see Figure 95.

![Figure 95: Point location on cell boundaries](image)

The algorithm sequence follows the right handed-orientation of the cell parametric coordinate system from left to right.

For 1D segment:

- $u < 0 \rightarrow$ left cell
- $u > 1 \rightarrow$ right cell

A point can be located in only one of these regions. In case of a triangle cell $T2N3$, there are 6 regions, see Figure 96, with the following constrains:

<table>
<thead>
<tr>
<th>Edge $e_i$</th>
<th>$e_0$</th>
<th>$e_1$</th>
<th>$e_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constrain $c_i$</td>
<td>$c_0 \equiv v &lt; 0$</td>
<td>$c_1 \equiv 1 - u \cdot v &lt; 0$</td>
<td>$c_2 \equiv u &lt; 0$</td>
</tr>
</tbody>
</table>

Each constrain reduces a possible point location to a half plane. As shown in Figure 96, there are 8 possible locations. The following proposition is a combination of constrains:

$$ (c_0 \land c_2) \Rightarrow c_1 $$

and when applied, it reduces the number of possible point locations to 7. The complete number of locations is shown in the truth Table 23, indicating true T or false F, when tested against boundary constrains of the prescribed regions. The region $x$ is not existing because the point would need to be located in $R_1$ and $R_2$ at the same time, which is impossible because the two regions do not overlap, see Figure 96.

<table>
<thead>
<tr>
<th>Regions</th>
<th>$x$</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>4</th>
<th>0</th>
<th>1</th>
<th>IN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_2$</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>$c_0$</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>$c_1$</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

*Table 23: Triangle truth table*
Constrains in Table 23 can define a point location inside or outside of the investigated cell. This is not sufficient for the evaluation of the ray exit edge/face and the calculation of intersection with exit edge/face. The orientation of the ray helps to identify more rapidly the intersection by introduction of the ray intersection constrains as follows:

Figure 96: Possible positions of the point leaving the cell in the neighborhood of the triangle cell
From the Table 24 we can find out the number of tests and the number of intersections with boundaries necessary to complete the identification of the ray exit and its intersection coordinates with the cell boundary. The most complex computation is required when the exit point is located in $R_5$ and $R_2$. In that case, three tests and two intersections must be performed.

For the quadrilateral cell $T2N4$, the following constrains are checked for the identification of an exit point location in the cell external regions, see Figure 97:

Each of these constrains reduce the possible point location to the half plane as indicated in Figure 91. There are 16 possible situations in the truth table, and the following propositions reduce this number to 9.

\[
\begin{align*}
c_0 & \Rightarrow \overline{c_2} \\
c_1 & \Rightarrow \overline{c_3}
\end{align*}
\]

In addition, to the boundary constrains the ray intersections constrains are introduced.

Table 26 after the elimination of impossible cases and identification of external cell regions defined in the truth Table 25.
<table>
<thead>
<tr>
<th>$R_i$</th>
<th>$c_i$</th>
<th>$e_j$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_0$</td>
<td>$c_0$</td>
<td>$e_0$</td>
<td>$r_i(e_j)$</td>
</tr>
<tr>
<td>$R_1$</td>
<td>$c_1$</td>
<td>$e_0$</td>
<td>$r_i(e_j)$</td>
</tr>
<tr>
<td>$R_2$</td>
<td>$c_2$</td>
<td>$e_0$</td>
<td>$r_i(e_j)$</td>
</tr>
</tbody>
</table>

**Table 26: Quadrilateral constrains path**

**Figure 97: Possible positions of the point leaving the cell in the neighborhood of the quadrilateral cell**
The next cell types are 3D cells. The simplest case is the tetrahedron which can be associated with quadrilateral because they have the same number of constrains:

Figure 98: Possible positions of the point leaving the cell in the neighborhood of the tetrahedron cell

Figure 99: Possible positions of the point leaving the cell in the neighborhood of the prism
Figure 100: Possible positions of the point leaving the cell in the neighborhood of the pyramid
Figure 101: Possible positions of the point leaving the cell in the neighborhood of the hexahedron
The Newton-Raphson method is used in a completion point extraction phase where a candidate cell and point are in the neighborhood of the point to be found. The point extraction algorithm in its first phase finds the candidate points in modeling coordinates $x$. The Newton-Raphson method computes in iterative loop a sequence of positions which lie increasingly close to the ultimate location. This can be presented as a multidimensional root-finding problem on a distance function $d$ that measures the error between the specified constant location $x$ and the guess solution $A(u)$, see section 1.3.1.

$$d(u) = A(u) - x \Rightarrow 0$$

1.4.4 - 8

The prerequisite to find the candidate point in the neighborhood of the proper minimum ensures that the subsequent refinement phase doesn’t inadvertently fall into an erroneous local minimum, as the field can have several non-zero minimum. The iteration is initialized from an arbitrary location inside the cell $u_0$ and is repeatedly shifted through a sequence of new locations $u_{i+1}$ which approach to the unknown location $u$. The Newton-Raphson method is defined with the recursive equation

$$u_{i+1} = u_i - \frac{d(u_i)}{d'(u_i)}$$

where

$$d'(u_i) = \frac{\partial A(u_i)}{\partial u_i} = J(u_i)$$

1.4.4 - 9

The iterative algorithm applying the equation 1.4.4 - 10 has to result in the quite small distance $d$ between successive points $u_i$. If the distance is increasing the point parametric coordinates are outside the range [0...1] and the analyzed point is not located in the investigated cell. For the calculation of the starting point (seed) in the parametric space the applied Newton-Raphson method uses the cell connectivity and the point location tests, which define in addition an exit cell side. Thus, the Newton-Raphson algorithm can be restarted with the new cell guess.

The identification of the starting cell is done applying the local value algorithm; see section 1.4.4, for finding the point on the surfaces and the cell containing the point. Because the surface is part of a 3D domain, the surface knows about the volumetric cell and edges defining that surface cell. Ones the surface cell is identified the volumetric cell is defined based on the surface cell index mapping, through which each surface cell has a link to the volume cell origin.

This algorithm utilizes the knowledge of the normalized cell boundaries aligned with parametric coordinate axis so the points are checked against the simple boundary conditions for example if the $(u,v,w)$ value is between $(0,1)$. 

111
An application of the Newton-Raphson point location algorithm is related to the usage of numerical probes tools when for example the user provides input for one of the following Representations: Local Value, Local Isoline or Vector Line, discussed later in the thesis, see section 2.4.3. All of them have in common that the point location in the parametric cell space needs to be computed.

Given a point in a physical space \((x,y,z)\), the problem is to find out if a investigated cell is containing or not this point. The algorithm has to perform the in-out checks applying the cell normalized coordinates \((u,v,w)\), and it turns out that this computation is one of the most computationally intensive task when the before mentioned representation are constructed. The objective is that such algorithm is efficient. Conceptually, the task involves the application of the Jacobian matrix of the isoparametric mapping, which provides only local information for each cell. The marching cell algorithm can be an applied for getting a correct cell. The optimization of the point location algorithm to define the starting seed cell for the marching cell algorithm is done by the traversal of the domain boundaries. After that, just the selected cells are taken into account. When these cells are analyzed the Jacobian matrix for the analyze cell is calculated and a center of the cell represents the initial point location guess.

\[
\begin{bmatrix}
\Delta u \\
\Delta v \\
\Delta w \\
\end{bmatrix}
= J^{-1}
\begin{bmatrix}
\Delta x \\
\Delta y \\
\Delta z \\
\end{bmatrix}
\]

\[
\Delta u = J^{-1} \Delta x
\]

\[
\begin{bmatrix}
\Delta x \\
\Delta y \\
\Delta z \\
\end{bmatrix}
= J
\begin{bmatrix}
u \\
v \\
v \\
\end{bmatrix}
\]

\[
J \Delta u = \Delta x
\]

Note that even within the current cell the metric terms vary, so that the algorithm is of iterative nature until \(\Delta u=\Delta v=\Delta w=0\). If \((u,v,w)\) are outside the range \([0,1]\) or depending of the cell type boundary constrain, the target point is outside the current cell and another neighbor cell will be analyzed.

Given an arbitrary point \(x\), and the good guess of the cell where the point is located, a Newton-Raphson procedure is used to calculated the corresponding \(u\). The initial guess \(u^{(0)}\) is taken to be a center of the cell and than we apply the previously mentioned equation, in an iterative algorithm as follows:

\[
u^{(i+1)} = u^{(i)} + \Delta u^{(i)}
\]

\[
u^{(i+1)} = u^{(i)} + J^{-1}(i) \Delta x^{(i)}
\]

If the connected cell is searched, the algorithm converges quadratically to the current point. If the cell being searched doesn’t contain the point, the new value of \(u\) will exceed the cell normalized space \([0,1]\) interval). The search is than switched to the neighbor cell in for \(u^{(i+1)}\). With the possibility to find the connected cell, the algorithm is repeated for the next connected cell, till the solution or boundary of the domain is reached. In the case that the point is not found, and there are no other connected cells because the boundary of the domain is reached, the algorithm looks for other seeds (boundary) cells and the marching cell algorithm continuous on the cells, which are still available for traversal. Here the problem is to identify the cell boundary through which the point moved out of the cell. The whole algorithm is situated in the normalized cell space where the boundaries are simplified, and nicely aligned with the coordinate axis planes. It is interesting to reuse the found point and cell information for the next point inclusion calculation, because the algorithm is often repeated for the point in the neighborhood of the previous one.
1.4.5 Particle Trace Algorithm

Scientific visualization tools for particle tracking apply algorithms, which difference comes from the calculations space (modeling or parametric) in which the integration is performed. If the calculation is made in the parametric space [86, 87] the particle trace points are found through the mapping coefficients, which relates parametric and modeling space. As these coefficients are computed in the parametric space, they imply the calculation of Jacobian matrices, which lower the algorithm performance, and in addition the particle trace points have to be mapped back from the parametric to the modeling space, in order to be displayed. This method is constrained to the irregularities of the modeling grid, which can introduce errors to the computed vector field transformations. A more direct method for calculating particle traces [88], is to use the modeling coordinates, but then the performance penalties are due to the difficulties in calculating the particle trace points location. The method based on generalized stream function for 3D [89] describes a steady flow applying two coincident scalar fields representing dual stream functions, where the particle trace is found as the intersection of these two iso-surfaces [90]. The limitation of this method is the coarse resolution of the vector line inside the cell, which is represented only as a straight line segment. More conventional approaches includes the adaptive step sizing and more efficient integration formulas based on Runge-Kutta or Adams-Bashforth schemes [91].

This section describes the particle trace algorithm [92], named also Vector Line algorithm in this thesis, and applied to the velocity fields defined on structured and unstructured meshes.

A vector line is an imaginary curve where a direction at any of its points is the direction of the vector field at these points. Vector lines never intersect, because a point in the vector field can have but one direction, only one line can pass through it. It is assumed that the vector field is stationary, continuous and single valued and that the particles are mass less, definitions follows:

- **Stationary** means that the vector field is independent of time. The trajectories of the particles are called streamlines.
- **Continuous** means that streamlines do not break up.
- **Single valued** means that the particle cannot split and occupy neither two places nor two distinct particles occupy the same place. Streamlines never intersect.

![Figure 102: Tangency condition for the vector line analytical and numerical treatment](image-url)
The mathematical concept of the particle motion is described by a point transformation of the particle position $x$ during time $t$, see Figure 102. Consider the point $P$ in a vector field $v$. The vector line through the point is a general 2D/3D curve. The position vector $x$ gives the location of $P$ as a function of some parameter $t$ that varies along the vector line. The tangent to the curve at $P$ determines the vector line

$$\frac{dx}{dt} = v(x(t))$$

1.4.5 - 1

The numerical integration takes into account the discrete points of the computational grid which defines the vector field $v$. Each vector line is a solution of the initial value problem governed by a vector field $v$ and an initial seed point $x_0$. This curve is depicted by a ordered set of points $(x_0, x_1, x_2 ... x_n)$ defined by

$$x_{i+1} = x + \sum_{i}^{t_{i+1}} v(x(t))dt$$

1.4.5 - 2

Adjacent points are connected and define the curve geometry which is displayed. The Vector Line representation can be applied to surface and volume vector fields defined in 3D space. Thus, there are two different vector line algorithms: one for the treatment of volume vector fields, and another for the treatment of surface vector fields. For example, the last one can be applied to cutting plane and isosurface vector fields. The flow in the surface is taking into account the projection of the tangential component of the velocity field. The following two sections treat these two aspects.

Volume Vector Line

The vector lines are the solutions $x(t)=[x(t),y(t),z(t)]$ of the system of ordinary differential equations, see equation 1.4.5 - 1. The value of the $v(x)$ is defined by the interpolation algorithm local to a cell, which contains the point $x$, where the vector field $v$ is described with a finite number of vectors $v_s$ given at cell nodes $x_s$. The applied interpolation algorithm is the isoparametric mapping, as developed in section 1.3.1 for each cell type, where the vector field at an arbitrary point inside a computational cell is obtained by the interpolation in the parametric space, and transformed back to the modeling space as follows:

$$x(u,v,w) = A[u(u,v,w)]$$

1.4.5 - 3

The isoparametric mapping operator $A$ defines the coordinate’s transformation of the vector line points from the parametric space $(u,v,w)$ to the modeling space $(x,y,z)$. The applied Runge-Kutta integration method [93], explained later on, on page 119, requires the parametric cell vector field $g(u,v,w)$ at the specified point inside the cell. The application of the equation 1.4.5 - 3 to the vector line’s equation 1.4.5 - 1 in the parametric cell space leads to the following result:

$$\frac{dA(u)}{dt} = v[A(u)]$$

1.4.5 - 4

which is equivalent to:

$$\frac{du}{dt} = g[u(t)]$$

1.4.5 - 5

With

$$g(u)=J^{-1}(u) \cdot v[A(u)]$$

1.4.5 - 6
and the Jacobian $J$ of the isoparametric mapping $A$:

$$J(u) = \frac{\partial A}{\partial u}$$

1.4.5 - 7

The equation 1.4.5 - 1 is computed with interpolated values from $g(u)$. The parametric vector field $g(u)$ is computed for each cell node and applied for the definition of the inverse mapping $J^{-1}$. The isoparametric algorithm is efficient when processing:

- the vector value at the point inside the cell, requires $J^{-1}$,
- the point inclusion test that defines if a point is located inside the cell, requires $A$.

The point inclusion test is efficient because the cell boundaries are planes or lines aligned with main coordinate axis of the cell parametric space, see section 1.1.1. The conditions that the mappings $A$ and $J^{-1}$ exists are:

- the vector field $v$ should be single-valued, i.e., when the mesh contains singular points (several different grid points occupy the same location in space) a numerical solution has to ensure that these points have identical values for $v$.
- The Jacobian matrix $J$ must be non-singular in order to be inverted.
- The mapping for the right hand side of equation 1.4.5 - 6 calculating $g(u)$ must guarantee enough continuity for the solution of $u(t)$ throughout the approximation of $J$.
- Continuity should be ensured across two cells. This is satisfied with the piece wise (cell-by-cell) isoparametric mapping. It must be noted that too distorted cells must be avoided.

Let $x_s$ be the cell nodes associated with vector field $v_s$ in modeling space, and let $U_i$ be the cell parametric space $(u,v,w)$, oriented according to the right-hand side rule so that the cell boundary normal point outwards the cell. The vector line algorithm consists of the following steps:

1. find seed cell $U_i$ for the given $(x_0,y_0,z_0)$
2. find seed point $(u_o,v_o,w_o)$, in $U_i$
3. define isoparametric mapping $A$ for $U_i$
4. define Jacobian $J_s$ for each of the cell nodes for the mapped cell vector field $g_s = (J^{-1})_s v_s$.
5. define isoparametric mapping $A^{-1}$ for $U_i$
6. integrate the vector field equation $g$ applying the isoparametric mapping $A^{-1}$. The integration is performed by a fourth order Runge-Kutta method in the parametric cell space $U_i$. The integration is continued until the vector line crosses a boundary of $U_i$.
7. find the intersection of the vector line $u$ with the cell boundary $U_i$. The intersection point becomes the last point of the vector line $u$ local to cell.
8. map the vector line $u$ to the modeling space $x=A(u)$.
9. find the neighboring connected cell $U_j$.

If the connected cell $U_j$ is found, reuse the intersection found in step 7 as the first point of the vector line and replace $U_i = U_j$. Repeat steps 3-9.

If the connected cell $U_j$ is not found the mesh boundary is reached and the vector line algorithm stops.
The algorithm described above allows computing a vector line starting from some initial position \( x_o \) in modeling space. However, it is common to compute the vector line which reaches a given point in modeling space. Usually, a vector line is computed over the complete mesh. For example, a trace that consists of the vector line that reaches point \( x_o \) together with the field line that starts from \( x_o \). To compute the field line that reaches \( x_o \) one simply computes a vector field line starting from \( x_o \) but with a minus sign in the equation 1.4.5 - 1:

\[
\frac{dx}{dt} = -v(x)(t)
\]

1.4.5 - 8

with solutions \( x(t) = (x(t), y(t), z(t)) \). The overall result of the vector line algorithm consists of two distinct segments of the vector field line, representing the forward and backward sweep respectively.

The vector line algorithm has following important issues:

- identification of seed cells and points
- integration step magnitude
- integration break
- mapping of parametric coordinates based on cell connectivity

Initial seed cell and seed point location are found with the point location algorithm, described in section 1.4.4. Given the coordinate \( x_s \) of the seed point, its corresponding coordinates in parametric space are found. To calculate \( u_0 (u_o, v_o, w_o) \) from \( x_s = A(u_0) \), the nonlinear system of equations is solved using Newton-Raphson’s method. Since the mapping \( A \) is isoparametric, the system of equations has only one solution and Newton-Raphson’s method, see page 111, can safely be used. The integration of the interpolated \( g \) within one cell \( U_i \) is performed by a fourth order Runge-Kutta method, see page 119. It is carried out in the \( (u,v,w) \) coordinate system from the first point in the new cell to where the integrated vector line point leaves the cell. An average velocity \( g \) is defined from velocities defined in cell vertices as

\[
|g| = \sum_{i=1}^{N} |g_i|
\]

1.4.5 - 9

The integration step \( \Delta t \) is calculated for each cell so that approximately \( M \) steps are taken in the cell. The \( M \) parameter can be interactively adjusted by the user then a step size \( \Delta t \) is given by:

\[
\Delta t = \frac{1}{M|g|}
\]

1.4.5 - 10

![Figure 103: The cell boundaries parametric space in 2D](image-url)
Figure 104: The cell boundaries parametric space in 3D
Experiments have confirmed good results with $M=4$ or $M=5$. The different reasons, for which the vector field line algorithm (page 114) has to be stopped, are called integration breaks, and they are activated when:

- vector line reaches the grid boundary (step 9)
- number of vector line points have been exceeded
- number of vector line points per cell have been exceeded
- The Jacobian matrix $J$ is singular (step 5), thus, the mesh contains a degenerate cell.

The point location and moving the parametric point from one cell to another is accomplished by applying the cell connectivity and the imposed orientation of the cell faces defined for each cell type, see Figure 103 and Figure 104. The objective is to avoid the computation of the point parametric coordinates ones the vector line points are mapped to the modeling space. As the vector line exit point is known applying the point location algorithm, the parametric coordinates of the point on the cell boundary can be viewed from both connected cell. The mapping is done in following steps as shown in the Figure 105 and Figure 106:

- map the cell parametric coordinates to the boundary, which results in lowering the parametric dimension.
- swap in orientation of parametric coordinates between connected boundaries
- map the cell boundary parametric coordinates to cell, which results in raising the parametric dimension.

The important precondition is that the cell topology for the whole domain is completely defined, as described in section 1.2.2.

The local parametric cell coordinates are transformed when passing the interface between two cells without the necessity to find the interface point in the global grid coordinates avoiding Newton-Raphson method to find the point location in $(x,y,z)$

**Figure 105: The map of a cell boundary point between connected cells in 3D**

**Figure 106: The map of a cell boundary point between connected cells in 2D**
The mapping is based on following steps:

\[ P(u, v) \quad I \text{ cell} \quad A(T(u_i, v_j)) \]

\[ T(u_1^i) \quad \text{interface } I \text{ cell} \]

\[ T(u_1^*) \quad \text{interface } II \text{ cell} \]

\[ B(T(u_2^*)) \]

\[ T(u_1, v_1) \quad II \text{ cell} \]

1. \( T(u_i) = A(T(u_i, v_i)) \)
2. \( T(u_i^*) = B(T(u_i^*)) \)
3. \( T(u_2, v_2) = C(T(u_2^*)) \)

The generic algorithm is based on \( A, B, C \) mapping, which are defined separately for each cell type involved. The solution provides the required flexibility to support heterogeneous cells without complicating the generality of the algorithm. The ability to change \( A, B, C \) at run time without the requirement to change the general algorithm and the ability to allow the programs to be fine turned after it is up and running (interactive turning).

### Table 27: The mapping procedure of a cell boundary point between connected cells in 3D

<table>
<thead>
<tr>
<th>3D scheme: ( T(u', v') )</th>
<th>( A(T(u_1, v_1, w_1)) )</th>
<th>( T(u', v') )</th>
<th>( B(T(u', v')) )</th>
<th>( T(u_2, v_2, w_2) )</th>
<th>( C(T(u_2^*, v_2)) )</th>
</tr>
</thead>
</table>

In the first phase the point must be defined in \((u, v)\) coordinates local to the face. This is handled with the \( A \) set of functions, see Table 27, where \( w \) coordinates are always zero as they are part of the cell faces. If that is fulfilled the intersection of the vector line with the cell edge end face is found. This algorithm includes the identification of cell edge, which is used to identify the next cell and thus maintain the \( C_0 \) continuation of the vector line. Each cell which is holding the vector line has at least one interior and exit point. This is just not true for the cells in which the vector field vanishes. For each cell type all the possible cases for entering and exiting of the vector line are considered as detailed in the section 1.4.4 related to the point location algorithm.

### Surface Vector Line

The surface vector line algorithm requires the same steps as the space vector line algorithm, in order to calculate the surface particle traces. What makes it different is that the volume vector fields are projected to the tangential plane of the surface points, which are defined for the created or inputted zones. The isoparametric mappings for surface cells are derived from section 1.3.1 where maps between 3D modeling and 2D parametric space require particular treatment, for the calculation of the inverse Jacobian matrix.

### Runge-Kutta Method

The integration of ordinary differential equations is solved by the Runge-Kutta method [93, 94] The vector field \( v \) is the RHS of the following equation:
\[
\frac{dx}{dt} = v(x, t).
\]

1.4.5 - 11

The idea is to rewrite \( dx \) and \( dt \) as finite steps \( \Delta x \) and \( \Delta t \) and multiply the equation by \( \Delta t \).

\[
\Delta x = v(x, t) \Delta t
\]

1.4.5 - 12

This is an algebraic equation for the change in \( x \) when the independent variable is stepped by one step size \( \Delta t \). In the limit to make step size very small a good approximation of the equation 1.4.5 - 11 is achieved. This explanation results in the Euler method

\[
x_{i+1} = x_i + v(x_i, t_i) \Delta t
\]

1.4.5 - 13

which advance the solution from \( t_i \) to \( t_{i+1} = t_i + \Delta t \) and if expanded to power series \( O(\Delta t^2) \) can be added to the equation 1.4.5 - 13. If we consider the trial step at the middle of the interval

\[
k_1 = \Delta t v(x_i, t_i)
\]

\[
k_2 = \Delta t v(x_i + \frac{k_1}{2} , t_i + \frac{\Delta t}{2})
\]

\[
x_{i+1} = x_i + k_2 + O(\Delta t^2)
\]

1.4.5 - 14

This is called the second-order Runge-Kutta method. As there are many ways to evaluate the right-hand side of the equation 1.4.5 - 12, which all agree to the first order but may have different coefficients of higher order error terms. With the Runge-Kutta method, the adding up the right combination of these coefficients the error terms are eliminated order by order. The fourth-order Runge-Kutta method is defined as follows:

\[
k_1 = \Delta t v(x_i, t_i)
\]

\[
k_2 = \Delta t v(x_i + \frac{k_1}{2} , t_i + \frac{\Delta t}{2})
\]

\[
k_3 = \Delta t v(x_i + \frac{k_2}{2} , t_i + \frac{\Delta t}{2})
\]

\[
k_4 = \Delta t v(x_i + k_3, t_i + \Delta t)
\]

It requires four evaluations of \( v \) per step \( \Delta t \). This is more efficient to the equation 1.4.5 - 14 if at least twice a large step is possible with

\[
x_{i+1} = x_i + \frac{1}{6}[k_1 + 2(k_2 + k_3) + k_4] + O(\Delta t^5)
\]

1.4.5 - 15

The higher order integration method is always superior to a lower order one, but when we consider the number of arithmetic operations involved and the possibility of an adjustable integration step to achieve a desirable accuracy, it is well known that the fourth-order Runge-Kutta method represents an optimize choice. This is the reason why the fourth-order Runge-Kutta method is used for the calculation of volume and surface particle traces algorithm, as described in this section 1.4.5.
2 Adaptation of Visualization Tools

Visualization tools for interactive quantitative and qualitative data analysis have to provide an effective and easy to use functionality for the examination of fluid flows and their underlying physics. The interactive visualization mimics the experiment in the laboratory. It requires to localize and to isolate regions where, for example, a shock interaction or vortex generation flow pattern is present. If several visualization tools “instruments” are applied together, they can intuitively stimulate new ways to analyze and discover fluid flow behaviors, especially valid in the regions where experimental measurements were not taken.

The Representations are the graphical objects, discussed later on in this chapter, which are appearing on the computer display and they are responsible for displaying geometry and different scalar and vector quantities. As the thesis investigates a real-time interactive approach, considerable attention is given to the design and development of numerical probes and their Graphical User Interface (GUI) setup. Although quite massive calculations are involved for each invocation of such visualization tools, the achieved interactive feedback loop between a user and the visualization system is quite satisfactory, even on today’s ordinary PC hardware equipped with a standard 3D graphics card.

During the visualization process, the user can restrict his/her analysis to a specific interactively selected Zone; see section 1.1.4, to which different quantity fields could be related. The number of introduced quantities is left open to a user input. In addition, representations are also available for the experimental data sets, which are commonly used for performing comparisons with computed data sets.

The numerical probes are interactive diagnostic tools supporting scalar and vector representations, for example:
- isolines and color contours for scalar fields
- smoke/dye injection simulation of particles with no mass
- bubble wire traces for velocity profiles

They are integrated in the visualization system interaction mechanism for providing information feedbacks between displayed information, loaded input parameters, selected algorithms and the manipulated graphical model. Some of the typical user interactions are:
- mouse driven positioning of the starting point selection
- menu driven selection of the numerical probe geometry (point, line or plane)
- interactive color map modifications
- mouse or menu driven selection of quantity threshold range,
- arbitrary positioning of a section and cutting plane

The user interaction is made transparent through the use of mouse and keyboard input devices. The examples of interactive user actions are the selection of an Active Surface or the manipulation of the Viewing Buttons, which control the setup of the user viewing position and perspective angles. These interactions are in the direct relation with the displayed geometries and quantity representations, and when applied they trigger an immediate visual feedback of the visualization system.

Two different groups of algorithms are applied when the graphical representations are generated:
1. the algorithms involved in geometrical search
2. the algorithms that generate the image

The first group of algorithms is explained in detail in chapter 1 and the basic layer of this second group of algorithms is today supported in hardware by the 3D graphics cards, which implement the OpenGL 3D API. The OpenGL low level graphics library is encapsulated in the Graphics category, explained in the implementation section 3.7.2.

The designed scientific visualization system manipulates structured and unstructured surfaces in a transparent way, enabling that the same types of graphics representations are provided for both types of geometries. The representations are classified in three main groups as geometry, scalars and vectors.
Geometry representations parameters and attributes are:
- predefined boundary types identification
- repetition: translation, rotation and mirroring,
- wireframe,
- hidden line/hidden surface removal,
- flat, Gouraud and Phong lighting interpolation algorithm,
- positioning of light sources,
- material properties of the surface (transparency, texture mappings)
- mesh quality assessment:
  • cell quality check
  • distorted cells based on edges or normal ratio criteria
  • small cells

Scalar representations are:
- color contours
- isolines
- scalar section and grid line distribution in the Cartesian x-y plot
- local value
- local profile
- traversal functionality for computational surfaces
- cutting plane
- iso-surface, clouds effects with transparency

Vector representations are:
- vector field
- vector section
- local profile
- cutting plane:
  • control of vector length: uniform or according to linear or log scale magnitude
  • uniform or colored according to a scalar quantity
- particle path:
  • color and line type option
  • graphical and numerical control of the released location

Beside the classical field representation of isolines, contour shadings and thresholds, the numerical probes as diagnostic tools, provide support for an interactive quantitative analysis of vector and scalar fields, and their related geometries. Numerical probes are adapted for the localized user investigation, aided with mouse point and click operation. The following numerical probes are designed:
- local values
- local isolines
- sections with quantity distribution along curves (arbitrary or mesh lines)
- local profiles
- local vector lines

Numerical probes are displayed as point, line or plane object, which are interactively controlled to investigate the displayed Zones. The objective is to restrict the analysis to the user selected zones, which are obviously not the only part of the complete calculated domain. For example, the isoline is created on a surface and displayed. Thus, only the isoline appears on the screen and the entire domain and reference surfaces involved in the isoline generation could be hidden by the user. In such way the user interactively defines the visualized scene content, and obtains focused, filtered and reduced set of displayed graphical information.
The local value probe displays a numerical value of the scalar field at the interactively selected point. The interactive section line input results in the Cartesian plot showing the quantity distribution along the created section. The local profile is a numerical probe which allows, to locally blow up a region, a direct example is the boundary layer, and shows the quantity distribution in a specially designed Cartesian plot representation, which is displayed at the location of the inputted section. As mentioned, this feature is valuable for the investigation of flows in boundary layers, as the geometry scale along the section can be adjusted independently of the viewing coordinate system in which the representation is posted. In Figure 107, local profiles are applied around the airfoil together with few particle paths. It can be noted that the vectors are not tangential to the particle paths as the local profiles show exploded region of the boundary layer.

All the previously mentioned probes are also available for 3D computations. In Figure 108 we can see the local value tool to display density values combined with local isolines, streamlines in the form of stream-ribbons especially adapted for 3D vorticity interrogations. The important approach in volume analysis of the structured data set is the volume slicing by extracting surfaces of constant $I$, $J$ or $K$ indices. The volume traversal enables to browse rapidly through the mesh and to localize interesting field features in an animated “surface” way. The geometrical mode allows the visualization of surface geometry, while in the quantity mode the shaded scalar contours are displayed. The possible switch between these two modes of interaction can be done without interrupting the visualization process.

For all types of 3D data the volume investigation is based on the isosurface and cutting plane numerical probe. The unstructured surface is the result in both cases. On that surface, other quantities can be interpolated and analyzed using the same numerical probes and their representations (isolines, color contours, etc.) as available for the surfaces provided by user input. The interactive user input for both tools is provided in numerical and interactive mode. For the cutting plane tool the scrolling and rotation about an arbitrary axis are provided, and for isosurface the interaction with the Colormap enables mouse click input of the isosurface scalar value.
By combining different types of representations on the screen the graphic output can have qualitative and quantitative meanings, as indicated for some representations in Table 28.

<table>
<thead>
<tr>
<th>Qualitative</th>
<th>Quantitative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream lines</td>
<td>Vector arrows</td>
</tr>
<tr>
<td>Isosurface</td>
<td>Isolines</td>
</tr>
<tr>
<td>Contours</td>
<td>Plots</td>
</tr>
</tbody>
</table>

*Table 28: Qualitative or Quantitative character of the Representations*

## 2.1 Input Data Model

The input data model describes the organization of a numerical data expected to be specified by a user. The user input consists of a mesh geometry and different number of quantities. The 1D, 2D or 3D mesh geometry is composed of different cell types; as described in the Data Modeling section 1.1. The basic assumption is that the mesh cells do not overlap. Each cell represents a unique definition of the approximated geometry space, it is covering. Usually, such requirement is assured by the grid generation system responsible for the grid modeling.

The input data model of the visualization system has to be adapted to different usages. For example, the imported geometry could be checked to be verified against the initial CAD model. Another example is, when data from a CFD flow simulation are combined with data from FEA stress computation and needs to be visualized in order to treat the fluid-structure interaction problems. In addition, measurements data could be provided and comparison against the computed results could be requested.

The data models for computational methods can be decomposed in two large groups, and such categorization is based on the way how the computed variables are related to grid cells or nodes:
1. The cell-centered input model defines the variables at cell centers.

2. The cell-vertex data model defines the variables at the cell nodes.

Because the output of each numerical simulation software is specific to its computational method (cell-centered or cell-vertex) it is appropriate to have a unified input model for a scientific visualization system. In this thesis, the cell-vertex model is chosen as the input data model, as it is more near to the data model of the applied graphics model. Additional reason for selecting the cell-vertex model was that it was not obvious how to extrapolate the input data to the domain boundaries when treating the cell-centered input model. The applied mapping for converting the cell-centered data to cell vertices is defined by averaging the quantity values of the neighbor cells surrounding each node, as explained in node topology, see section 1.2.4.

The designed input data model supports the multi-block decomposition, which is especially useful for the treatment of complex geometry.

The quantity input data types are organized as:

- field data
- solid data
- validation data
- plot data

Field data are quantities given for the complete computational domain. Solid data are quantities confined to solid boundaries, such as the heat transfer and skin friction coefficient. Validation data are coming from experiments or other computations to facilitate the data comparison. Plot data are related to the arbitrary x-y plots, such as convergence history. An arbitrary number of quantities are allowed for each data type.

The results of the computational analysis can be mapped to the designed input file system, which consists of a combination of ASCII and binary files responsible storing different input data sets. The input file system is organized through a small navigation file, which groups all the files related to the specific project. Different navigation files can be created for the same data sets giving flexibility in organizing the visualization session. The navigation file defines geometry dimension, topology type, number of domains and number of quantities for each data type. It defines global transformation of overall geometry like mirror and repetition. For each domain, the number of nodes and cells are given together with domain connectivity file indicating the types of defined boundaries, as specified in section 1.2.5.

The quantity files are related to the defined quantity data set specified per domain or per solid surfaces. Therefore, they are separated into two categories: field and solid data files.

The field data files are defined for the complete domain. For example, the pressure or the velocity field around the space shuttle is defined for each grid node.

The solid data are restricted only to the solid boundary segments. For example, the skin friction on the solid surfaces of the space shuttle can represent a valid solid data input. Another distinction is made between the computational data and the validation data. For example: comparisons have to be made between experiments and the computed data, see Figure 109 where the pull down menu reflects the mentioned distinction. In Figure 110, for the airfoil example, the field data pull right menu contains a list of scalar and vector quantities defined in the computational space the solid data are also present under similar pull-right menus, defined only the solid boundaries.
The Validation Data menu for the airfoil example is shown in Figure 111. It contains all the validation data for the project as specified in the input file.

The Plot Data is the fourth and the last defined quantity type. The selection of such data type allows creation of simple Cartesian plot with arbitrary number of curves. The example is again from the same airfoil computation showing convergence history, see Figure 112. This data type is introduced to give more possibilities for presentations of data that are not directly linked to the project.

The mesh topology classified as structured or unstructured, requires different input files. The structured grid topology has all mesh points lying on the intersection of two (or three) families of curves, which define the curvilinear coordinate lines. They are represented by a set of integers as $i,j$ or $i,j,k$ depending upon the space dimension $2D$ or $3D$ respectively, see Figure 113.
The *unstructured grid topology* is formed by combination of different cell types, see *Figure 114*, where the grid points cannot be identified with coordinate lines, but have to be indexed individually in a certain order. It requires a more complex bookkeeping with respect to structured grids, because cell connectivity has to be indexed separately for each cell-nodes relationship, while the structured cell connectivity can be explicitly used from the grid indices.

In the input model the domains connectivity, see section 1.2.5, is defined and in addition the boundary conditions are specified. The orientation of domain boundaries is determined and the connectivity rules are applied for the matching neighboring segments in order that the consistent input setup is established. The boundary conditions (BC) are defined on domain boundaries and depending on the project dimension associated to the 2D or 3D segments. This information is mainly used for the following purposes:

- when initializing the visualization session, by default the solid surfaces are displayed, as they represent naturally the investigated problem (airfoil, airplane),
- they can be identified and selected in this starting phase to get a first insight in the available data,
- The periodic and connected boundary conditions are applied when particle trace algorithm is performed for the multiblock data model.
The standard notation for the description of boundary conditions, see Table 29, is applied on each segment separately. Since the types of boundary conditions may vary according to the used flow model, a common set of BC types was defined for the input data model, as follows:

<table>
<thead>
<tr>
<th>BC types</th>
<th>Input Abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>CON</td>
</tr>
<tr>
<td>Periodic</td>
<td>PER</td>
</tr>
<tr>
<td>External</td>
<td>EXT</td>
</tr>
<tr>
<td>Singularity</td>
<td>SNG</td>
</tr>
<tr>
<td>Inlet</td>
<td>INL</td>
</tr>
<tr>
<td>Solid</td>
<td>SOL</td>
</tr>
<tr>
<td>Outlet</td>
<td>OUT</td>
</tr>
<tr>
<td>Mirror</td>
<td>MIR</td>
</tr>
</tbody>
</table>

*Table 29: Standard notation for boundary conditions*

Defining the domains connectivity is a time consuming task, as the user needs to specify the relative positions of the interfacing Segments. Normally this information is created in the grid generation phase [95], where the connectivity of the boundary elements is specified along these interfaces according to the processed domain topology taking into account their orientations, see section 1.2.5. The orientation parameter refers to the relative position of grid points on two connected Segments (equal or reverse parametric coordinate direction along the same arc length). Thus, for 2D grids, the connectivity of two boundary Segments (curves) is specified by indicating the segment orientation. For 3D grids, the parametric local coordinate system and the orientation of BC segments (surfaces) need to be specified. The segment corners nodes represent the possible locations at which the origin of a parametric coordinate system could be located (considering corner 1 as the reference on the first patch: 1-1, 1-2, 1-3 and 1-4). Thus, the segment connectivity can be specified in eight different ways. The connected segment indices are also indicated using the standard notation, see section 1.2.2.

*Figure 115: Domains connectivity for structured 3D multiblock grids*
2.2 Surface model

The Surface model is a central organizational concept of the visualization system, as it defines a reference base to access the created and displayed Representations in a Scene. The surface model is extended with the representation container, which enables the surface model to keep track of the created representations, and to incrementally associate the surface data during the visualization process. For example, the Geometry menu offers the set of possible Actions as: select remove, create and delete the surface. Following the same idea, other geometry models as points, curves and bodies are treated identically. The following explanations are based on the Surface model and the related Representations. Each surface has two possible states:

**Active:** when its related representations can be invoked, such as geometry rendering or scalar fields contours, and

**Inactive:** when a surface is present in the Scene, but the invoked representations are not applied to it.

In order to improve the user awareness of the active or inactive state of displayed surfaces, the user can select the surface by pointing to it with the cursor. The mouse click will trigger the visualization system to localize the surface and inform the user of its state. This feature enables the user to apply representations selectively to each displayed surface. Such interactive way of modeling the scene by controlling the created representations for each surface offers a large set of combinations to create appealing graphical presentations.

The surface solid boundaries are the initial representations displayed when the visualization is initialized in order to provide an intuitive feeling for the inputted geometry, see Figure 116. These representations are depending on the project dimension:

- **for 2D:** all boundaries defined in the boundary conditions data set.
- **for 3D:** only solid boundaries, if they don’t exist, all the boundaries defined in the boundary conditions data set.

The Geometry/Surface menu, see Figure 117, allows the creation, removal or deletion of surfaces. The existing surfaces are organized in two groups. The first group includes all the existing and created surfaces in the project and this list is only one. The second group of surfaces is defined for each View and these surfaces are building the View scene. This mechanism allows that each surface can be selected.
and reused in a multi-views visualization environment. However, the \textit{surface-representation-view} relationship keeps the displayed representations independent from each other. An operation performed on a surface in an active view will not affect the same surface representations in the other views, thus the surface representations may be different for each view.

The visualization tool for the extraction of surfaces from the structured topologies is making use of \textit{I}, \textit{J} or \textit{K} mesh indices. The tool is invoked by the \textit{Geometry/Surface/Create} dialog-box; see Figure 118, which contains:

- the surface name if saved, thus we create it
- the selection of \textit{I}, \textit{J} or \textit{K} surface index
- the start index value of the surface to be first displayed
- the Min-Max range of index values to display a part of the surface (patch)
- two different scrolling modes
- the \textit{Save} and \textit{Cancel} operation
- the multiblock mode for selecting individual domains

The dialog-box allows the traversal of computational domains by scrolling through constant \textit{I}, \textit{J} or \textit{K} surfaces and if desired, to save these for further analysis. For the multidomain configuration, only the selected domains are traversed. The traversal consists of displaying the surfaces in two modes: animated or step-by-step. The displayed representations are:

- \textit{Geometry}: as a surface grid
- \textit{Quantity}: as shaded color contour, which is appropriate for quick localization of interesting 3D regions

![Figure 119: Structured multiblock surface patch manipulation](image)
Figure 119 shows on the left side the Ahmed body multiple patches of the structured grid surfaces and the respective scalar field contours and on the right side the Hermes space vehicle discretized with the structured 2-block grid. The interesting aspect in both cases is the space traversal, which is performed with displaying complete or partial surfaces; in addition, indicating the multidomain grid structure for geometry and quantity representations. The possibility to create surface patches is defined within Min-Max range of the local $I, J$ surface indices. For example, the surface extraction can be reduced to cover a limited display area in order to make other representations visible, as shown in Figure 119, where the surface patch was adapted to show the particle traces in the background. The minimum surface patch size can be reduced to one cell. When the surface or surface patch is created the surface boundaries are displayed, as part of the interactive system feedback, to indicate that the surface creation was successful. A counterpart of such interactive visualization tool does not exist for unstructured meshes.

The cutting plane and isosurface tools are the interactive tools, which are available for both structured and unstructured topologies. It is important to mention that the related surface extraction process is involving all the domains of a multidomain (multiblock) input. The created surfaces are automatically named with the domain index and prefix, such as ISO for isosurfaces and CUT for cutting planes, thus preserving the relationship with the domain from which they were extracted. In Figure 120 the example of simultaneous application of both tools is shown on structured and unstructured meshes.

The Selection Surface dialog-box contains the surfaces associated with the view. As shown in Figure 121 the created isosurface and cutting plane are highlighted as active and the automatic naming convention is applied, as described in the previous section. The interactive process to remove or destroy the surface is equal to the surface selection process. When the surface is destroyed all associated representations are removed in all the views. If the surfaces are visible on the screen, the interactive surface selection can be performed with the mouse point and click operation. Every visible surface in the view can be made active or inactive depending on its previous state. The interactively selected surfaces are highlighted and the repeated action on the same surface will act as an active/inactive toggle. For improving the interaction feedback, the surface name and the active state are displayed in the monitoring area of the main window. In addition, the Selection Surface dialog-box is helpful when reusing the same surfaces in a multi view visualization environment.
2.3 Geometry Representations

The different geometry representations are depicted in the geometry pull-down menu, see Figure 122, as follows: Full mesh showing the surface grid, Mesh boundary showing the surface boundary in 2D and boundary surfaces in 3D. Solid Boundary automatically displays only the boundaries/surfaces with the imposed solid boundary condition. The geometry representations are unified for structured and unstructured grids, as the basic graphics for both topologies is defined by their nodes, edges and cells, see Figure 123.
Some geometry representations are based on the wireframe graphics composed of polylines, and others representations are constructed of filled polygons reflecting the surface cells structure. The boundary and solid boundary representations are used for fast identification of overall geometries as they are the lightest graphics models. The Repetition functionality enables the duplication of the displayed geometry, such as: the translation in a given direction, the rotation about and arbitrary axis or a mirror operation. The Figure 124 shows the three possible examples of the geometry repetition. The rotation about an arbitrary axis and the translation of geometry are especially useful for turbomachinery applications, where the blade to blade geometry can be duplicated for the better understanding of the flow quantities behavior in the regions with periodic boundary conditions.

The Render pull-down menu allows the display of the active surfaces with Hidden line, Flat, Gouraud or Phong lighting interpolation, see Figure 125. The hidden line representation assigns a uniform color to all the surface area. The algorithm removes the invisible parts of the surfaces.

The hidden line algorithm is combined with grid wireframe representations to display the cells shapes. If the flat rendering is applied the surface color is modified in accordance with the light position and cell surface normal preserving the uniform color inside the cells. The Gouraud and Phong lighting interpolation technique made the scene more realistic because the lighting interpolation is performed inside each cell according to light position and cell vertex normal. Figure 126 shows all the four rendering possibilities for the space shuttle.
2.4 Quantity Representations

The scalar and vector representation are treated both as representations and based on the selected quantity the menu items are adjusted automatically, see Figure 127. The Representation menu is a content sensitive menu, as its content depends on the type of the selected active quantity (scalar or vector). This is similar to the geometry, where the Geometry menu depends on the input data dimension (2D or 3D) and grid topology (structured or unstructured), see section 2.2.

![Figure 127: Scalar and vector representation menus](image)

The Scalar Representations are applied to the whole surface field as Isolines and Color Contours. Other extracted representations are geometrical subspaces of the surface like the section (curve) and the local values (point) types. The quantity representation becomes something which can be graphically displayed as point-marker, curve-edge, surface-face or other textual information. All these graphics shape primitives are used extensively in combination with colors. It is important to stress that setting of the color, as one of the fundamental graphical attributes, has to be performed easily, especially in the case when interactively adapting the colors mapping.

The Vector Representations are modeled as glyphs, in different directional arrow based shapes, defined with color and shape parameters, which are associated respectively to the vector magnitude and direction. Such vector glyphs, or icons, are scaled by a set of predefined values, which can be interactively manipulated. The two important representations are the Vector Field representation which displays the vectors glyphs at predefined surface points, and the Vector Line representation graphics displayed as curve. The application of such vector representations depends directly on the analyzed fluid flow. For example, the Vector Field representation is not appropriate to be applied where the grid points are dense and the vector magnitudes change rapidly. In such case, the displayed vector arrows will fill up the displayed area and the presented graphics will be more disturbing than helpful. In order to have better visibility of the underlying flow pattern the vectors distribution needs to be coarser than the grid distribution, thus it is suggested that the limited number of Sections or Local Vectors are interactively placed in the combination with particle traces starting in the problematic region, for example, where the vortex core is located.

In this section, each Quantity representation will be tackled in a combination with other available interactive tools, in order to indicate the best practice in how to use them in an appropriate and an efficient manner. As the visualization system offers the possibility to combine several representations in the same view, in many of the described examples their simultaneous usage is pointed out in order to explain the potential added value of such visualization scenarios.
2.4.1 Isolines

The Isolines interactive tool offers several options to create isolines:
- isolines are computed within specified minimum-maximum range
- giving the increment between two adjoin isolines
- specifying the number of isolines

In Figure 128 the Isolines are displayed together with Local Value representations, in order to enhance the relationship of displayed graphics with numerical values.

The Isolines can be generated as a representation group using the described dialog-box or by the Local Isoline tool, for the creation of individual isolines. This is performed by specifying a scalar value or by locating the surface point, through which the isoline is requested to pass. For both invocations the parameters can be set interactively or numerically. The first approach enables interactive input, while the numerical one assures that the exact numerical input of an isoline value or surface point coordinates are given. For the interactive approach, the creation of an isoline by the mouse-click operation is done by pointing with the screen cursor inside the colormap scale or by pointing the cursor over an active surface. The mouse click will trigger the interpolations: of the selected value or of the selected point location, followed by the isoline generation algorithm for the identified scalar value.

2.4.2 Quantity Fields and Thresholds

The Quantity Fields representation group brings together the scalar color contours and vector field’s representations, as they put on view all the inputted surface data. For each surface node the mapping functions between the color and the quantity value are defined. The scalar contours are continuously coloring the surface cells, while in the case of vector field representation the vector magnitude and color parameters are applied for the vector glyph generation. The extension to such full field surface visualization is the partial display of the quantity field specified through the threshold values.
The Color Contour representation is the most frequently used representation used in case to gain an overall behavior view of the scalar field. Each surface vertex is associated to a color which corresponds to a scalar value at that point. The scalar field is painted with a graduation of colors according to the selected color mapping displayed through the Colormap, see Figure 131.

The coloring options are reassembling the ones used for the Geometry Rendering where just one surface color is applied to different rendering methods. In this case the number of colors is increased from one to the number of colors used in the applied colormap. These options are equivalent to the geometrical ones and activated through similar menu items, for example the ones with flat and smooth labels. The Flat Contour displays the scalar field as collections of cells where a unique color is assigned to each cell. The assigned color corresponds to the average value of the quantity in that cell. The Smooth Contour displays the scalar field as collections of vertices where inside each cell the color is smoothly changing from one cell vertex to another. In this type of representations the cell geometry is not visible. To see the cell wireframe geometry together with smooth contours the geometry wireframe representation can be superimposed to it, see Figure 132.
The extension to the surface Color Contour representation is the Threshold representation, which restricts the coloring of the surface to a surface region depict by the Threshold range, see Figure 132. In the case of an airfoil only the supersonic scalar values of a Mach number above 1 are displayed with the Threshold Color Contours representation. In the subsonic region the Isolines representation is applied, and in the entire field the quantitative information is displayed with some Local Values representations.

The Vector Field representation draws the vector quantity at the grid nodes of the selected surfaces. In the case of an unstructured grid the vectors are displayed at each grid node. The structured grid topology gives to the user the possibility to display the Vector Field at different resolutions by defining the numbers of vectors to be displayed in the $I$ and $J$ directions, see Figure 133.

In Figure 134 the examples of unstructured and structured vector field’s representations are presented. It can be noted that the resolution of the structured field can be controlled with limiting the number of display vectors, thus making more comprehensive the presentation of the examined vector field.

The Threshold tool provides a convenient way to discard “uninteresting” parts of the vector field. The surface regions in which the vectors magnitude is outside the threshold are discarded. Figure 135 shows the threshold representation in two cross sections of the airplane wing where we isolated the vectors near the Mach 1 velocity field.
2.4.3 Point based numerical probes

The point based numerical probes allow interrogating scalar and vector fields at user’s defined points. The outputs are the Local Scalar, Local Isoline and Local Vector representations. For improving the user positioning within the viewing space and to verify his/her point locations input the Coordinate Axis tool provide the necessary visual validation feedback, see Figure 136. The interactive approach supported with the point based tool allows the use of the mouse as the point input device, or a string input, if exact numerical coordinates values are requested. The selected point in the displayed view space doesn’t need to be a surface point. The considered line is perpendicular to the screen and passes through the selected point, thus the line defined with the cursor point and the view normal will be applied in the intersection algorithm, as explained in section 1.4.4.

In the Figure 136, for the velocity vector field the quantity range is adjusted using the Range menu, see Figure 137, between 350 and 430 m/s, and we generate some Local Isolines representations with a precise numerical input.

In addition, the isolines are validated with the Local Scalar values corresponding to the velocity magnitudes. Finally, we have added the Local Vector representations, to show the correspondence with the newly created Colormap.

The Range menu makes possible to modify the current range of the quantity or to reset it to the default state. This option is useful for the comparison purposes between different projects, in order to synchronize the imposed applied ranges. The range limits can be interactively changed utilizing the string input or the mouse point and click operation within the Colormap display area. The user interaction is identical to the threshold range input described in the previous section. The new setup up of the quantity range will affect the representation associated with the related colormap, and consequently all the dependent representations will be updated inline with the new color mapping. In Figure 136, it is visible that the colors of the isolines and vectors are in accordance with the range indicated with the displayed Colormap in the right side of the figure.
2.4.4 Curve based numerical probes

There are three basic types of Representations coupled with the curve based probes. These curve based representations are applying the geometries defined through:

1. predefined curves
2. sections (in addition, Cartesian plot for scalar fields) Isolines, Particle paths
3. and Local Profiles

The common curve representation is the extraction of structured curves based on the $I, J, K$ indexing, including the surface boundaries which can be further refined, filtering out only the parts with solid boundary condition, as shown in Figure 138. The Cartesian plot representation, see Figure 139, is complex representations as it relates two views and their respective curve representations: 1) in the main view the curve shape is displayed and 2) in the Cartesian plot view the quantity distribution is shown.

When the calculation of the scalar distribution along the extracted curves is called the view, called Cartesian plot view is opened. The limits of the plot axis are automatically sized to the min-max limits of the quantity value calculated from the sections present in the Cartesian plot. The Cartesian plot layout can be modified by the Coordinate Axis Editor. For the Cartesian plot view, the view manipulations buttons can be used to adjust the displayed space of curves geometry and scalar quantity range. For example, the zoom area operation can be used to blow up a select a region of a Cartesian plot.

Figure 138: Cartesian Plot menu

![Figure 138: Cartesian Plot menu](image)

Figure 139: Scalar distribution along solid boundaries and sections

![Figure 139: Scalar distribution along solid boundaries and sections](image)
In Figure 139 are shown 2D Solid Boundary and Section representations for three different scalar quantities. For the two plots, from left to right, the abscissa represents the \(x\) coordinate and the third plot is related to Section representations where the quantity distribution is presented with arc length starting from the solid boundary. A Section representation displays the geometry of the curve as a result of plane/surface intersection, see section 1.4.1. The Section input requests two points for defining section location. It is based on a mouse-cursor visual feedback. After the input of the first section point is made, the red rubber band line attached to the cursor, visually aids a user to define the section. The second mouse click triggers the section creation. In addition, to this input type, a numerical input with precise section coordinates is possible through string based field using keyboard. A special option is the input of a vertical section, which is defined by one point and the view-up vector. Its generation is triggered by a mouse double click action at the cursor location. This approach avoids the input error of two consecutive clicks at the same location, as in such case the section is not defined, and by default the vertical section is made at that place.

The Grid Line distribution is a specific representation for structured topologies. A special dialog-box, see Figure 140 in the upper left corner, is created to handle the interactive aspect for such selection. It can be noticed that the black surface curve is the grid line, which is almost following the shock wave and has the highest Mach number distribution as shown above in the Cartesian plot. The difference between the Apply and Accept button is that Apply button is used to traverse the grid lines for temporary viewing, while Accept assign the curve to the Cartesian plot. The next two plots in Figure 140 are related to 3D computation, and show the Section tool applied to the Cutting Plane instance, created bellow the double ellipsoid. The coordinate system representations help the user in orienting herself/himself within the 3D space and in addition, they verify that the sections are made, in this example, along the \(x\)-axis. In 3D the section line input is the cutting plane, which is perpendicular to the computer screen. The found intersections between of the cutting plane and the active surfaces appear as 3D curves on each intersected surface. Their related scalar distributions are mapped to the Cartesian plot.

Figure 141 shows an example of a grid line distribution. To create such representation the user selects an appropriate constant grid line index, commonly \(I\) or \(J\), used for structured surfaces created from \(I, J\) or \(K\) constant indices. The \(I\) and \(J\) indices appear in the dialog-box shown in the lower left corner, and have to be interpreted in accordance with the rules explained in the boundary condition definition, see section 1.2.5.

The counterpart of the scalar distribution plots along curves or section are the vectors representations along the constant \(I\) or \(J\) grid lines, with also the possibility to extract only part of an identified curve. The designed dialog-box is equal to the scalar counterpart and defines the same functionality. The Vector Filed Grid Line and Section distributions are shown also in Figure 141.
The section oriented input is also used for creating a set of Isolines or Particle traces, as shown in Figure 142. The section span is breakdown in number of reference points through which the isolines or particle traces have to pass. On the ship hull a set of three groups of isolines are created for the bulb area showing the pressure field, while in the upper part three groups of Surface Particle traces are showing the shape of the wave generated by the ship. What is interesting to notice is that by making the ship hull transparent, we have still the feeling of the ship surface, while having the possibility to see the hidden part of the mirrored representations.

The algorithm behind the Section representation is based on the marching cell algorithm; see section 1.4.1, which calculates the vectors at the intersected points of a surface with the plane, which is defined by the active view normal and the inputted section. The same algorithm is used for the Local Profile representation. The difference is that the vector representations do not need an additional Cartesian Plot to view the extracted scalars or vectors. The Local Profile representation modifies the coordinate space metrics by allowing that the Local Profile coordinate axes are stretchable, thus adjustable by the user. This characteristic allows the "magnifying" or "blow up" effect of the investigated region resulting in a special Cartesian plot representation, which is inserted in the active view at the inputted section location.
Thus the Local Profile behaves like a virtual microscope, which is very appreciated by the users investigating the boundary layer flow patterns. Figure 143 shows an example of the normalized velocity U component compared with the validation data coming from experiments. The inputted line section localizes the region to be mapped to the Cartesian plot representation plotted in the view itself. The validation data, as shown in the example, can be combined with the local profile representation.

The user interaction is equal to the Section input, thus a red rubber band line appears as the visual feedback for the profile positioning. The coordinate system is created based on the following assumptions:

- When the two points are inside the active surface, the position of the origin is defined with the first selected point.
- If one of the points lies outside the surface boundary it is reset to the intersection point with the surface boundary. The intersected point becomes the origin of the coordinate system.
- If both points lies outside the boundary while the section passes through the surface, the point which has the lowest y coordinate between the two intersections becomes the origin.

The quantity axis in the analyzed region is associated with the abscissa. It is drawn perpendicularly to the y axis. The quantity axis parameters are modified through the Quantity Axis editor shown on the left side in Figure 143. The analyzed distance is associated with ordinate axis. It starts from the origin of the coordinate system and ends at the selected points. The distance axis is modified through the Analyzed Distance editor shown on the right side in Figure 143, and it is divided into three sections for setting: the Analyzed distance, the Axis length and the Plotting parameters. The Analyzed distance section modifies the real analyzed distance as follows:

- **Full Section** option extends the analyzed region along the whole mesh. It is similar to the Cartesian plot representation, but is locally plotted on the view.
- **Between selected points** option analyzes the region between the two selected points, eventually resets them to the mesh limits. This is the default representation.
- **On a given distance** option requests the analyzed distance in user units. This item is essential as it allows inspecting very small regions inside the boundary layers. If the given distance is larger than the full section distance, it is reset to the maximum limit.

![Figure 143: Local profile representation for boundary layer analysis](image-url)
The Axis length section modifies the y axis length. It is the distance along which the distribution is plotted:

- **Not magnified** option draws the axis from the origin and ends it at the analyzed distance. In this case no blow up is performed and the quantity distribution is drawn in real scale.

- **Between selected points** option draws the distance axis between the two selected points. If the first point lies outside the boundary limits, it will be reset to the closest boundary intersection with the section line.

- **In centimeters** option requests the y axis length in cm to be printed for example on A4 page.

- **Analyzed x d** option requests the magnify value. The y axis length is given by the product of the analyzed distance and the magnify value. In this way the user can specify the number of times the analyzed distance is magnified.

The plotting options section customizes the Cartesian plot scales and appearance. The Vector local profile is used in the same way as the scalar local profile, with the exception that the quantity axis does not exist in this representation, see Figure 144. What is interesting to note is that the particle paths are not tangential to the vectors, which is correct, as the geometrical spaces differs.

### 2.4.5 Surface based numerical probes

Surface based numerical probes are the **Cutting Plane** probe, which extracts planar geometries and the **Isosurface** probe, which extracts geometries with constant scalar value. It is evident that both tools need 3D geometry and quantity field to be defined. As indicated, the **Isosurface** tool is only applicable to scalar fields, thus it cannot be applied to vector fields, while the Cutting plane is operational for both types of quantities. The interaction with the cutting plane tool is provided through the dialog-box, see Figure 145.
The cutting plane probe provides functionality to slice the 3D computational field in three different modes, as indicate in the dialog-box: Polygon, Geometry and Quantity. The Polygon mode orients the cutting plane with a simple solid based visual feedback within the viewed 3D space. The Geometry mode is more complex as it performs the intersection algorithm throughout the full mesh geometry. The Polygon mode is computationally the most demanding, as after performing the intersection algorithms, it needs to interpolate quantity fields’ values for the new cutting plane and to finish create the smooth color contour, as explained in section 2.4.2. In addition, the cutting plane dialog-box offers additional functionality for interactive plane positioning, as rotation and translation buttons related to the stepping parameter for the controlled slicing of the 3D quantity fields. The important button is the Save button which save the cutting plane for further analysis. After the cutting plane is saved, it becomes available for interaction like all the other surfaces, see section 2.2. Figure 146 shows an arbitrary cutting plane of the pressure field around the F-16 airplane. There are three cutting planes created as constant X, Y and Z planes in which the pressure Contour Thresholds are defined with predefined Colormap range to make also visible the plane solid shape with the pressure Isolines. The solid surface transparency was applied in order that the volume perception is improved. Similar effect was achieved by displaying the geometry in the light gray color, of the previously saved X constant cutting plane, visible in the background. The same example was used for the vector Sections in the cockpit area in the Y constant cutting plane, see Figure 147.
Figure 148: Several isosurface representations of the Temperature field around the airplane

The Iso-Surface probe is the second tool for the inspection of the 3D volumetric scalar analysis. The isosurface calculation is based on the marching cube algorithm; see section 1.4.1, which result is an unstructured surface for which the prescribed scalar value is constant. Figure 148, shows an example of the Temperature field isosurfaces surfaces around an airplane, where temperature maxima are located in the region of the engine inlet and outlet. The interesting aspect in this figure is the cut out of the isosurfaces in the mirror plane in order to see the temperature field development. On the right side of the picture, with the use of transparency and gradual removal of the airplane solid body, the complete shape of the temperature distribution could be analyzed. The interaction with the Isosurface probe is supported by two menu items: one for creation and the other for saving of the generated isosurface. The isosurface scalar value input is entered through the string input field, while the interactive mouse based input is to move the cursor inside a colormap and with the mouse click select the desired scalar value, which will trigger the creation of the isosurface. The isosurface algorithm involves the traversal of the complete computational data input, thus it is computationally intensive for large 3D meshes. Ones the interesting isosurface is found it can be saved for further manipulations, similar to the cutting plane, and become one of the other active surfaces available for interactive manipulation. In Figure 149 the combined usage of different probes is presented.
2.4.6 Vector line numerical probes

The vector line numerical probes generate particle paths by integration algorithm in the vector quantity field, see section 1.4.5, and obviously do not have their counterpart for the scalar field’s analysis. The computed particle trajectories are modeled as 2D or 3D polylines. As shown in Figure 150, the interactive process is controlled through the Vector line pull-right menu, which contains the Local Vector Line, Local Section and Grid Line options. Once selected, they enable the user to enter the position of the starting (seed) point as the input to the particle trace algorithm. The additional required parameters are specified through the Parameters dialog-box, also shown in Figure 150. The user interactively inputs the particle path seed location using the mouse to point the cursor to the desired location and the clicking triggers the particle path generation. In addition, the exact numerical coordinates of the desired point can be inputted through String Input field. The explained point selection is the same, as for the Local Value, Local Isoline or Local Vector representation. As in the case of these point based tools, to speed up the input, the Section tool is applied to generate a set of seed points evenly distributed between the two inputted section points. The interactive visual feedback is the rubber band line. The difference in the user interaction is that the number of desired seed points can be modified through the String Input after each execution. For example in Figure 150, each group of colored vector lines is generated from the Vector Line Section tool. In the same Figure 150, the central dialog-box is used for setting the seed from the surface grid points, which is only valid for structured topologies because the grid points are linked to $I$ and $J$ indices, as explained in the definition of structured topologies, see section 1.2.2.

The Vector Line Parameters dialog-box controls the algorithm input (number of global points or number of the cell points, when computing the particle trace) and the appearance of displayed trajectories like the line color and thickness. In the 3D computations there are two types of vector line algorithms which can be invoked, see section 1.4.5. Their main difference is that they a confining the vector line to lie on the surface or to be positioned inside the volume. The surface vector lines are sometimes associated with the skin-friction phenomena, as shown in Figure 150 on the surface of the car or along the turbine hub. There is also useful to draw streamlines in a turbine axial cross-section, as shown nicely the tip vortex location, in the same figure.

Figure 150: Vector Line menus and representations
In Figure 151, the both kind of streamlines are generated for the airflow around the airplane. It is interesting to notice that the combination of Local Vectors and Surface Streamlines representations show clearly the region of the wing tip vortex, in a cross-section plane perpendicular to the airplane flight direction. In right side figure, apart from the standard 3D streamlines strips, it is visible their intersection with a cutting plane instance, from where they were started. The Vector lines strips where created with the Mono setup, where a unique color is assigned to the group of vector lines. The Variable color option allows automatic drawing of vector lines, each with different color. Vector lines applying such coloring scheme improve the visibility of the swirl flow behavior, as shown in Figure 150 for the rear part of the car. The parameters for the control of the particle path computation are introduced because there is a possibility that the vector field generates, for example, cyclic streamlines which computation will never end. To avoid such problems the maximum number of points to be calculated inside each cell is set by the Cell field. To control the number of integration steps inside each cell the Cell average parameter guess the number of particle movements to traverse a cell. It is assumed that the cell traversal will be done, if the particle performs the prescribed number of steps, with the average cell velocity. The integration direction is shown in the Vector Parameters dialog-box, see Figure 150. In Figure 152, the Tk/Tcl CFView GUI from Numeca is shown. The red-blue vector lines are created with the downstream integration, while the yellow and violet with the backward integration from the structured surface nodes. The related dialog-boxes, for the generation of the Vector lines representations which aid in the interactive setup, are shown.

Figure 152: Vector lines representations from structured surface points, with the required toolbox in action
2.5 User-Centered Approach

The development of an interactive visualization environment has to consider cognitive ergonomics or human factor when designing the human-computer interaction (HCI) model. The cognitive ergonomics take care of the user mental processes of perception, memory, reasoning, and reaction aspects, tied to the envisaged interfaces and interaction techniques, required to be usable and receptive to the user's needs when he/her interacts with the visualization system. The interactive environment of a scientific visualization system must have a simple user-friendly Graphical User Interface (GUI) having in mind the following reasons: first to be considered, later on to become useful, and finally to ensure that the end users (CFD engineers or scientists) have an effective software workspace to work with.

The end user requirements are the key element in modeling the GUI of a visualization system, and they need to be continuously refined for each software development cycle. When the visualization system reaches its operational state, the users' observations and suggestions are considered with highest priority. Such software development approach is user-centered as the user directly participates in a development of the GUI model. When applied iteratively, it makes users contributions effective, and helps the developers to focus on the pertinent software features by filtering the good design ideas from the bed ones. The important outcome of such approach is that the involved users are satisfied, as they can experience that their input is continuously taken into consideration. The user’s requirements specificaion has to include enough information to enable the software developer to propose few solutions for the required functionality in order to have a better feeling for the envisaged interaction to be designed. This impose an additional constrain to the adopted designed methodology, as the identified user’s requirements has to checked for consistency. In the procedure-oriented classical approach, the required functionality is well known from the initial development phase, as the input and output operations are static and remain unchanged during the software development. In the interaction-oriented user-centered approach the system functionality is expected to change in an incremental and evolutionary manner. Thus the scientific visualization system design has to be adapted for a fast prototyping, allowing that the continuous usability tests are performed in order that the GUI model becomes as intuitive as possible.

<table>
<thead>
<tr>
<th>Classical approach/procedural systems</th>
<th>User-centered approach/interactive systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>formal problem analysis</td>
<td>analysis of possible user interactions</td>
</tr>
<tr>
<td>algorithmic development</td>
<td>development of interactive environment</td>
</tr>
<tr>
<td>adding user interface</td>
<td>adding algorithms supporting functionality</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 30: Comparison of classical and user-centered approach

The design of an interactive system implies the shifts in concepts: from data to visual information processing, from formal analysis to interactive modeling, from algorithmic to direct manipulation, from problem solvers to context sensitive environment and from passive support systems to intelligent cooperative system, see Table 30.

In the previous chapter the visualization tools and their possibilities were presented, and some interactive aspects of the visualization process were mentioned. The interaction modeling is specially related to the described numerical probes, which require an extensive and highly interactive GUI, in order to be used efficiently. The following list of requirements is considered, when the interactive model of the visualization system is designed:

- The visualization system has to provide efficient handling for large and heterogeneous data sets in a multi window environment.
• It has to be 3D graphics enabled in order to provide interactive an efficient: cursor control, visual feedback and status monitoring.
• The visualization tools and numerical probes have to perform fast in order to assure acceptable interaction response for the variety of visualization tasks, see chapter 2, for example when Isosurfaces or Streamlines need to be calculated and visualized.
• The transparency in manipulating data 2D and/or 3D, structured and/or unstructured, scalars and/or vectors, original input and/or derived quantities.
• The variety of input data formats for 2D and/or 3D, structured and/or unstructured, multiblock decomposition and connectivity specification, cell-vertex and cell-centered solutions with arbitrary number of input quantities.
• The variety of output data formats for generating image formats as bitmap, postscript and any other graphics file format output.
• A macro subsystem has to automate the user interaction.
• A help subsystem has to assist the user to use it correctly.
• Integrated databases access to remote data sources.

The designed interactive model supports the user’s active involvement in visual processing tasks. The main idea underlying the interactive visualization activity consists in letting the system reacting in a concrete and flexible manner to the user’s interactive request. The requirement is that the user controls the visualization experiment in a quick and productive manner, and that the graphical output of the investigated data takes advantage of the graphics acceleration hardware, adapted to the interactive manipulation of 3D models. The graphical presentations have to be compatible with the user mental representations and have to allow the communication and exchange of such results with minimum effort. If the system functions are not transparent to the user, the user will either reject the system or he/she will use it incorrectly. This is always an underestimated objective in the interactive system development planning, as commonly, the software designers apply the procedure-oriented approach, where the major role is the identification of a well-defined problem around a set of supporting algorithms.

The interactivity model is the central principle in the user-centered approach, as it establishes the system dynamics and offers possibility to support variety of user’s scenarios, by combining a finite set of available interactive components. The objective is to create intuitive visual human-computer interactions, which follows the specified concepts and operations to be given to the user and, at the same time that they are adapted to a specific user knowledge and experience in treating the simulated problem. The interactive environment has to support effective man-machine cooperation, mainly based on the software capability to complement the user activity with suitable feedback, as follows:

• the recalculation mechanism, which operates on the functional relationship between data introduced by the user and the application itself,
• the model consistency check, which guarantees system robustness at every stage of interaction,
• the immediate visual monitoring of the user actions, with useful confirmation indications.

When using an interactive visualization system, the end user can perceive the existing software objects. The clear example is the visualization system GUI composed of many different interaction tools, where each of these components can be invoked and manipulated separately. Thus each of them might be identified as Object, for the end user “thing” to employ and for the developer “thing” to develop. This relationship made possible a focused interaction between the two experts groups, as Object represents a tangible reality for both of them, users and
developers. The both groups work on a same object, but they are doing it from two different points of views. The added value in this process is that both groups are contributing to the development of the same object. Such approach is essential to the Object-Oriented Methodology (OOM) and tightly relates the user-centered GUI design with Object-Oriented Programming (OOP).

An interaction modeling is an important activity in a design process of a scientific visualization system. In the following section the applied modeling principles are described, followed by sections, which specify the details of interaction behavior (dynamic) and interaction data (static) aspects.

2.6 Interaction Modeling

The objective is to establish a set of common abstractions that allows the modeling of an interaction process. The adopted modeling approach covers the user’s interaction on the displayed objects and the dynamic modification of the related objects parameters. Their design is based on the following characteristics:

- the cognitive characteristics must improve the user intuition for specific activity,
- the perceptual characteristics applies color, depth, perspective and motion to improve the visual feedback,
- the ergonomic characteristics are concerned with the system usability (easy-to-use) and the learning phase through feedback monitoring of user actions and help facilities.

The effective use of GUI relies on the intellectual, perceptual and sensorial activities of the user. The GUI design requires consultation with cognitive scientist, psychologist, ergonomic expert, graphics designer, artist and application expert to work together in order to understand the complexity of the user’s tasks and how the user reacts when performing them. The interaction process needs to be translated into the specification describing a requested interaction: what the user has to do, and what the system has to respond.

The first phase in GUI design is to learn how the user thinks about the task and how he expects the work to be done (cognitive issue). The understanding of the user thinking process is the most difficult part during the first phase of GUI design. It consists of several trials-and-errors cycles, which are usually carried out with a set of small prototypes.

In the second phase (perceptual issue) the integrated prototype is constructed in order to find an appropriate GUI that supports the user-centered model, as the integrated GUI invokes functions, gives instructions, and allows control, and present results without intruding in the visualization process.

The adoption of the system is linked to the third phase (ergonomic issues) that enhances the user-system interaction space. The elements are the on-line context-sensitive help and the good hard copy print output possibilities. The main objective is to aid the user in developing the workflow for his/her visualization task to and to keep him/her aware of the involved data sets and applied algorithms.

The entities, which model the GUI architecture, are: Object, Event, Place, Time, State and Action. See Figure 153, where the ERM diagram of the interaction process is shown. In this diagram there are three important relationships:

2. Object is defined with its State and Place.
3. Object can perform Action, resulting in Object (Action) and Action can be performed Object, resulting in Action (Object).
The user Action produces an Event that triggers a system Action. The link between System and User Action is Event. Each user Action invocation must be uniquely interpreted by the system at every point of the interactive process.

The analysis of the user requirements, applying the user-centered approach, showed that a multi-window environment with menus and mouse controls covers most of the user interaction functionality and ease the access to the application. In order that the user can performed interactions with the system, the following generic types of user actions are considered to be triggered by the selected input devices, as:

- **menus** (mouse selection)
- **direct manipulation** (mouse picking)
- **key binding** (keyboard)
- **macros** (file)

In Figure 154, the menu structure is presented hierarchically in order to keep organized the actions around the similar content. The menu organization follows, from left-to-right, the visualization process: The **Project menu** is concerned with data input and output data formats. **Geometry menu** contains the mesh representation, followed by different Rendering possibilities. Than the **Quantity menu** selects scalar or vector field quantity, and the **Representation** menu offers creation of different representations possibilities. The **View** and **Update** menu are related to the setup of the viewing and presentation parameters. However, the menu structure is flat, which means that a respective menu items can be invoked at any level of interaction. The pull down and pull right menus, together with dialog-boxes enable the user to localize quickly the specific input.

However, the command sequence has certain amount of intelligence build into it. Only the commands that are consistent with the previous one will be executed. In addition, some commands can be triggered through the mouse-cursor point and click interaction; as sometimes, it is easier that the user performs a direct mouse manipulation than that he/her has to traverse all the menus hierarchy to find the envisaged command to execute. For example to select a surface, it is easier to do it by the mouse picking than to remember the textual name, which needs to be selected trough the dialog-box interface. The user’s control of the application by menus or by mouse has to result in a consistent visual feedback through input or output self-explanatory status monitoring. The GUI should provide a user-friendly point-and-click interface based on the use of mouse and keyboard, having access to variety of sliders, buttons, text entries, dialog boxes and menus for a variety of user’s inputs.
As mentioned in several occasions, the interactive procedures need to be ergonomic and effective. The fundamental user request for an interactive process is the user ability to instruct the application with point-and-click mouse actions and that the application reacts quickly by updating its display. A mouse movement is connected with the cursor position on the screen. By moving the mouse across the GUI layout the user activates menu items, selects active points and graphical objects. When the mouse is moved, the area that contains the cursor becomes active and all the mouse actions are affecting this selection area, for example a View.
The developed interface of CFView, see Figure 155, contains GUI components organized with menus, icons and dialogue boxes. Viewing operations and interactive interrogations of the computational fields are ensured by the cursor and view manipulation buttons. The general GUI layout is subdivided into different areas, as shown in Figure 155:

- MENU BAR area, containing TIME, DATE, MEMORY
- TOOLBAR area
- QUICK ACCESS area
- the middle GRAPHICS AREA and
- the bottom area, subdivided into following regions:
  1. message area
  2. string input
  3. viewing buttons
  4. view monitor
  5. cursor monitor

All the mentioned areas are 2D GUI components, except the graphics area, which is the part of the screen, where the graphics objects appear. The graphics area displays and manipulates the 3D graphics content, which is managed through specialized graphics objects called Views. One or more views may appear simultaneously within the graphics area. These views can be positioned arbitrarily, thus, they can be moved, sized and can overlap each other, see Figure 156. The generation of more than one type of view was analyzed and the generation of three types of views is adopted:

- 2D views that are primarily related to 2D projects,
- 3D views which are related to 3D projects and
- Plot views that are used to display data in a Cartesian plot form.

Figure 156: Different view types
This GUI model was presented at the AIAA conference in 1992 [96], and at time it was the first of that kind. The history showed that still today, see Figure 157, seems to be an appropriate and efficient GUI model for scientific visualization software; the industrial example is the Numeca CFView software [14]. It is interesting to note that the GUI was ported to variety of UNIX platforms and its successful reimplementation in Tk/Tcl by Numeca made it even more portable to the Windows and UNIX platforms.

We continue with the explanation of the GUI model, which components are controlling the graphics area and its multi-view structure.
Each view is allowed to have a different active state displayed in the view monitor. The views are graphic windows which enable the user to rotate, zoom and pan using a mouse. The use of multiple views with corresponding status displayed by the view monitor enhances the presentation of complex geometry and problematical phenomena. The user is allowed to create additional views by opening a new file or by explicitly requesting a new view for the same data. Special case is a Cartesian plot view, which is created whenever a new quantity distribution is requested. Depending on the view type, some menu items and viewing operations may be disabled (i.e., $X$ and $Y$ rotations are not allowed for 2D views). As visible in the Figure 156, some menu items in the top menu bar are grayed, which means that for the selected view, only the normally displayed items are active. This fact indicates that the menu system is content sensitive, which means that it adapts its content to the current situation of a user interaction.

A menu item invocation can be also handled by typing the key bindings assigned to the menu items and editing options starting from the menu bar, while the graphics area and viewing buttons have its own mouse bindings. The menu choices have keyboard short-cuts that allow experience users to quickly invoke different commands. In addition the menu items contain reminders that come with the menu item name, as indicated in Figure 158.

2.7 Viewing Space and Navigation

Although the scientific visualization system allows any number of views to appear in the graphics area, only the view selected by a user can be manipulated. Such view is named Active View and the user interactions are associated to it. The convenient way to make a view an Active View is to select it with the mouse point-and-click operation. The user positions the cursor on the top of any view and then by clicking the mouse button the view becomes active. The view is placed on the top of the view stack that contains all the views present in the graphics area. The newly selected view becomes the Active View and all the menus and viewing operations become related to it. The Active View is recognized by red color highlighting its border, see Figure 156.

The viewing space is represented as a cube, which is aligned with the Cartesian coordinate system. Another cube encompasses all the graphical objects present in the view scene. This cube defines the fitting space and it is used for the following reason:
• to evaluate the original view, when the view is created,
• to fit the graphical objects to the maximum viewing space, without changing the view orientation,
• to reset the view orientation, whenever the user has “lost” the content of the view.

Whenever a new graphical object, for example a surface is added to a view, a view automatically sizes the fitting space in order that all the graphical objects present in the view are enclosed in it, see Figure 159. The original view orientation defines the orientation of the view coordinate system according to the fitting space and the view limits. These view limits define the portion of the view space which is displayed, called a view scene. The original view displays all the graphical objects with the default view orientation.

All graphic objects lying outside the viewing space are clipped away, thus not displayed. Two clipping planes, the front and back clipping planes have an important application in 3D. These planes are parallel to the plane of the screen; and they delimit the viewing space in front and in the rear of the graphical objects that are clipped away. Parts of the objects may become invisible if these planes are very close to the graphical objects. This feature is very interesting for looking inside an opaque graphical object. The user can control this feature through a view button that allows moving these planes continuously forwards or backwards, as shown in Figure 160.

Figure 159: Cube model for sizing the viewing space

Figure 160: Clipping planes in viewing space
When moving the cursor in 3D, it is not always evident, which interactive point a user gives as input, as the cursor is not indicating a point but a line perpendicular to the display screen. In order to resolve such situation, an appropriate way is to display the location of a 3D selected point with the coordinate lines and the sphere marker, see Figure 161, which visually defines an Active Point representation. The Active Point is by default constraint to be inside the viewing space, as there is nothing to be selected outside it. In Figure 161 the viewing space is described with the coordinate system displayed in black and in addition, there is a viewing coordinate system, which is helping the user to stay oriented when navigating in 3D space. The possibility to input the exact coordinates is provided with a simple dialog-box, shown in the same figure, which is also reactive to the cursor position. This approach is also used during the interactive picking on a surface. In this case, a line passing through the active point and perpendicular to the screen is defined and used to find its intersection with the surface. As indicated in the above explanation a view can be decorated with more types of coordinate systems. The global coordinate system, has its origin always located at the lowest corner of the fitting space and is graduated, while the other one is the local coordinate system and has its origin at the current active point and is displayed without graduations. It defines the X, Y and Z rotation axis for the viewing operations. Another usage of the Active Point is to help the setting of the local coordinate system around which the viewing operations are defined.

A view can be provided by two types of projections for displaying three-dimensional objects: orthogonal and perspective (see figure 2). Orthogonal projections are typically used to represent the metric properties of an object and to show the exact shape of a side parallel to the plane of the screen. The perspective projection gives a more realistic representation of an object as seen from an observer at a specific position. This position, called the reference point, can be used to change the appearance of the graphical objects on the screen. The objects will appear more distorted when the point is placed closer to them. On the other hand, if the reference point is far from the objects, the projection will look like an orthogonal projection.
A viewing buttons provide a convenient way to interactively navigate in the 3D space and they are part of the GUI model, see Figure 163.

Figure 162: View projection types

X, Y and Z are the projection buttons that allow setting the camera position with setting a view normal parallel to X, Y or Z coordinates axis directions. The mouse-click operation triggers the setup action, also related to the last two buttons, the Fit Button which fits the scene to the view and affects the camera position parameters: target, width and height, see Figure 164, and the Origin button which brings the active view in its default viewing state updating all the camera parameters. The other buttons interaction is associated with mouse press and drag operations, and it modifies interactively the camera parameters in relation to the mouse movement, which is proportional to the mouse dragging direction and amplitude. The Scrolling Button allows translating the camera in a given direction, while the camera direction remains unchanged. Follows the Dynamic Viewing Button, which allows to rotate and translate the camera position around the target as well as to fix the rotation center, which model is shown in Figure 165.

Figure 164: Camera model and its viewing space
Rotation Buttons allow rotating the camera about the principal coordinate directions \(X\), \(Y\) or \(Z\) for 3D views, but it is deactivated for Cartesian Plots views. The Roll Button allows rolling the camera around the view normal and affects the view up vector direction. The Zoom In/Out Button allows to interactively zooming in and out, affecting the camera width and height parameters. The Zoom Area Button allows specifying a rectangular area of the active view to fit to the whole view display area and affects the camera position, target, width and height of the camera parameters, as shown in Figure 165.

Figure 165: Camera parameters and virtual sphere used for camera rotation

### 2.8 Visualization Scenarios

The visualization scenario is the set of interactive operations performed by the user, when creating visualization representations for selected quantity fields. The generated display layout is arranged and presented with a set of user defined views, realized through a kind of procedural template, usually based on standardized, common, or scientifically accepted output formats. The presented GUI supports such multi-view configuration, which allows simultaneous display and manipulation of heterogeneous data sets during the same visualization session. This feature is particularly useful for comparative data analysis when presenting information through standardized visualization layouts. In addition, the possibility to present different quantity fields in the same viewing space facilitates the treatment of multidisciplinary problems, where for example the fluid-structure interaction is analyzed.

The important issue to achieve this objective is the possibility to scale the geometry in order to unify the applied units (meter, feet) between different projects. The same approach has to be adopted for the investigated quantities in order that the quantity units are given in the same unit’s space.

The unification of the content and the application of same visualization tools, still do not provide that the displayed information is fully comparable. The use of the viewing parameters with a set of editors to adjust the appearance of the displayed graphics makes possible these final adjustments in order that the displayed data are equally processed. ones we have defined a visualization scenario with the layout configuration, with the unified data sets, with the same visualization representations, with the same viewing parameters and with the same graphics equalizing the decoration and appearance styles, we are in the position to look to different numerical simulations in a standardize way.
For such purposes, the Symbolic Calculator is an important element of the visualization system which can involve available quantities to define an algebraic expression, as shown in Figure 166. The Derived quantities are calculated from the user defined symbolic expression, and they can be field derived quantities (defined in the whole computational domain) or surface derived quantities (defined only on surfaces). Based on a standard set of computed quantities: static pressure, static temperature, density, absolute velocity vector field or relative velocity vector field; different new common Thermodynamical Quantities can be computed, like Mach number, Total Pressure or Internal Energy. In addition, differential operators, such as Gradient, Divergence or Curl, can be applied to the existing field quantities, thus resulting in the creation of a new scalar or vector quantity field, depending on the resulting quantity type resulting from each of the available differential operators, see section 1.3.2.

This process can be automated with a set of macro instructions, which can be invoked through the Macro subsystem. The Macro subsystem allows the user to record the interactive actions he/her is performing, in order that later on or next time the user is accessing the file, to automatically reply the performed action. This ability increases the user efficiency to investigate similar cases without repeating all over again the same set of actions. Thus, a user is actually becoming a high level programmer, and the macro script is the visualization programs, which can be used to define different Visualization scenarios.

In the following two examples, see Figure 167, of the standardized output created for the EUROVAL project [97], the two compared contributions are from Deutsche Airbus and Dornier. The airfoil computations were validated against the computational data. The grid mesh and pressure field are presented for the whole 2D flow field around the airfoil, while pressure coefficient, skin friction and displacement thickness distribution along solid boundary are presented as three Cartesian plots, validated against experimental data. As there were many European partners contributing, the generated visualization scenario macro has automated the generation of such presentations. The second example is the visualization scenario for 2D bump test case, as shown in Figure 168, where the vectors profiles in the boundary layer and the shock wave details with isoline and local values are presented. The two contributed computations are from University of Manchester (UMIST) and Vrije Universiteit Brussel (VUB).
Figure 167: EUROVAL visualization scenario for the airfoil test case
Figure 168: EUROVAL visualization scenario for the Delery and ONERA bump
As we can realize, the variety of presentation possibilities offered by the visualization system is large, and it is necessary to carefully experiment with available possibilities to come up with an appropriate setup. As it was explained, once the visualization scenario is found the user interaction with the system can be quite fast and straightforward, and even automated, for fast outputs as shown for the two EUROVAL examples.

The graphical primitive’s appearances influence the user visual perception of the displayed data. In Figure 169, the setting of Node in blue, Edge in red and Face primitive to be transparent, with gray grid lines for reference are shown. For comparison purpose, the use of these attributes can help to distinguish different data sets, when presented in the same view.

Additional possibility is the superposing of different views in a transparent mode, which gives an impression of one view. Such layering mechanism is interesting, as allows generating independent views, and then like putting transparencies one over other the comparison can be performed, as shown in Figure 170.
Modifying the colors associated to the scalar quantity can sometimes reveal new insight when analyzing the scalar fields. Together with the graphical primitives, where each of them can have different colormap associated, represents a powerful way to present data. This makes possible to manipulate in the same view different quantities, still keeping the necessary visual distinction for keeping them separate.

Figure 171: Different graphical primitives showing the same scalar field
Figure 172: Different colormap of the same scalar field
The power element of the Symbolic calculator is to generate new geometries, which are analytical shapes like sphere, and others. In the Figure 173, some examples are generated and the available quantity fields can be analyzed in these curvilinear spaces, as the superset of the cutting plane mechanism.

Figure 173: Analytical surfaces generation for comparison purposes

All the mentioned possibilities make the visualization system a versatile tool for comparison purposes.
3 Object-Oriented Software Development

This Chapter describes how the Object-Oriented Methodology (OOM) was applied to develop a scientific visualization system at all phases of the software engineering life-cycle, from problem statement, to analysis, design, implementation and testing/validation. In contrast to traditional methodologies, like structured systems analysis and design methodology [98], where functional decomposition is the primary concern and data are identified later, OOM focuses on the objects and their related data, then on the algorithms built around them. The present work demonstrates that OOM provides an approach to software development that those stemming from the traditional methodologies, especially for building software that is expected to evolve, as is the case for SV systems. The power of OOM becomes evident when facing complex applications which are tricky to develop using the functional analysis approach. Yet, any structured description of a problem has value, and users are often casting their views in functional terms (functional requirements). The role of the software developer is then to integrate these functional requests into the object-oriented model/specification.

3.1 Software Engineering Model

The building of complex systems -- such as scientific visualization systems -- is made possible by breaking down the whole problem into smaller, tractable and manageable problems that can be worked out by the developer. A model is an abstract construct that formalizes the design of a system and provides a sound foundation for implementing it. Several models can be used to represent different views of a system; complementary models pursue a specific purpose and are used to capture the crucial aspects of the system of interest.

In the software analysis phase we identify and depict the essential aspects of the application, leaving out all implementation issues. The analysis model is application-domain oriented: it includes descriptions of the objects tangible to the user. The application-domain model is an input to the design model, where objects will be designed for the selected computer platform, without consideration of coding and programming aspects. Finally, the design model will be implemented in a code written in a selected programming language (in our case, in C++).

ANALYSIS

The analysis begins with the problem statement expressed by the user.

Problem Statement: The analysis model is a concise, precise abstraction of what the desired system must do, not of how it will do it. The analysis model is to be understood, reviewed and agreed upon by application domain experts (who are not computer-science experts/programmers). This process leads to the extension of the user model data.

DESIGN

System design: System designs include system partitioning into subsystems.

Object design: Object design augments the content of the analysis model. Design decisions include: specifying algorithms, assigning functionality to objects, introducing internal objects to avoid re-computation, and optimization. The emphasis is on essential object properties, so as to force the developer to construct cleaner, more generic and re-usable objects.

IMPLEMENTATION

The overall system architecture defined by the design model is a tradeoff between the analytical model and the target computer platform. Some classes whose properties do not derive from the real world -- for instance Set or
Vectors which support specific algorithms -- are introduced as an auxiliary part in the design model. The implementation style must enhance readability, reusability and maintainability of the source code.

The models and the source code constitute together the software solution; they are the answer to the question **WHY is the software system created?** When developing a visualization system, several types of OOM models and diagrams are used to address three basic questions:

1. **WHAT?**

   The static model describes objects and their relationships by entity-relationship diagrams, a visual representation of the objects in a system: their identity, their relationships to other objects, their attributes, and their operations. The static model provides a reference framework into which to place the dynamic model and the functional model. The object model describes classes arranged into hierarchies that share common structures and behaviors. Classes define the attributes and the operations which each object has and performs/undergoes.

2. **WHEN?**

   The dynamic model describes the interactive and control aspects of the system by state diagrams. This model describes the time-dependent system characteristics and the sequences of operations regardless of the nature or mechanism of the operation. Actions in the state diagram correspond to functions in the functional model. Events in the state diagram become class operations in the object model.

3. **HOW?**

   The functional model describes data transformations in the system by data flow diagrams. The functional model captures the system’s functionality. Functions are invoked as actions in the dynamic model and are shown as operations on objects in the object model.

The static model represents the reference base, because it describes what is changing or transforming before describing when and/or how changes are done. Successful software engineering requires a number of very different technical skills to satisfy research and industry needs. They include the ability to do analysis, system design, programming design, coding, integration, testing and maintenance. OOM requires self-consistency and sense of purpose. The experience with OOM is not only based on a set of techniques but also on their interactions. That means that all of them work well together - much akin to the mathematical identification of simplicity and beauty. In order to use various notations and techniques in a balanced way, OOM focuses to achieve elegant design and implementation, which outcome is expected to be a comprehensive, implement-able, efficient, maintainable and extendable code.

Software engineering is the compilation of different activities, which have to follow and overlap each other in the cyclic and iterative manner. The class design should be a separate activity from the class implementation. Thinking only of a class design in an abstract manner can lead to a design that is impossible to implement. The balancing of these two thoughts is to the responsibility of an OO software engineer. The iterative process of the methodology improvement and the iterative process of software development are advancing in parallel. Consequently, the developed methodology extends beyond a single software project. Therefore it is not always possible to determine the effects of methodology crafting at the starts of the project, as the methodology evolves over the time in response to the acquired experience and this process is usually accompanies with development problems.
The software engineering methodology is an integrated combination of concepts, guidelines, steps and deliverables in the context of an underlying process description, includes not only graphical notations but also textual descriptions and documentation standards. The methodology encompasses a large integrated set of development techniques and tools resulting in procedures as

- **Debugging codes and algorithms to support development,**
- **Simulation results for industrial applications,**
- **New visualization patterns to accomplish research.**

Techniques are developed to improve software analysis and design possibilities:

- **Management techniques** comprehends planning, organizational structure (hierarchy of abstraction levels), deliverables (their description and timing) and quality control.
- **Design techniques** provide means for verifying the design before coding.

The library management provides support to the OO approach by search and query possibilities of existing reusable classes, when developing new classes. Tools supporting OO methodology provide notation, browsing and annotation capabilities. Ideally, the OO design tools should provide mechanisms for navigation from higher level OO diagrams to code and back to the design. This functionality is called forward and reverse software engineering relating the developed code with design diagrams. Software engineering must improve both: software quality (the product) and software production (the process). As B. Mayer says [4], there exists different quality factors, as follows:

<table>
<thead>
<tr>
<th><strong>Quality Factor</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Correctness</strong></td>
<td>is the ability of software products to exactly perform their tasks, as defined by requirements and specification.</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td>is the ability of software products to function even in abnormal conditions.</td>
</tr>
<tr>
<td><strong>Extensibility</strong></td>
<td>is the ease with which software products may be adapted to changes of specification.</td>
</tr>
<tr>
<td><strong>Reusability</strong></td>
<td>is the ability of software products to be reused, in whole or in part, for a new application.</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td>is the ease with which software products may be combined with others.</td>
</tr>
<tr>
<td><strong>Efficiency</strong></td>
<td>is the skilful use of hardware resources, processors, external and internal memories and communication devices, minimizing the resources and improving the performance.</td>
</tr>
<tr>
<td><strong>Portability</strong></td>
<td>is the ease with which products may be transferred to various hardware and software platforms.</td>
</tr>
<tr>
<td><strong>Verifiability</strong></td>
<td>is the ease of preparing acceptance procedures, particularly test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases.</td>
</tr>
<tr>
<td><strong>Ease of Use</strong></td>
<td>is the ease of learning how to use software systems, operating them, preparing input data, interpreting results and recovering from usage errors.</td>
</tr>
</tbody>
</table>

*Table 31: Software quality factors*
3.2 Object Oriented Concepts

The software development based on Object Oriented Methodology OOM stimulates the software designer to declare what a selected software component has to do, while the application of software components assure how the expected functionality is/will be accomplished. The software design defines a collection of software components – the ‘objects’ --, each of which encapsulates a specific part of the designed know-how to be implemented. When some of the needed software components are available, the task of the designer is considerably facilitated, since he/she can incorporate them in his/her design. Before starting implementing objects, one must analyze all objects and make a trade-off between ones that exist and the ones that need to be implemented.

Object modeling in OOM is expressed through the object’s declaration and definition specification, which provides the level of data abstraction and encapsulation necessary to be defined for considering the object reuse [99, 100]. The designer’s skill is revealed in the way he/she re-uses existing objects in combination with the newly-created ones that need to be developed to suit the software design specification. Re-using (validated) software elements is one of the fundamental principles/benefits of OOM: this contributes to improving the quality of the software by reducing the amount of programming and the risk of errors. Indeed, objects which are re-used are validated in many independent application contexts and continuously improved. Creating software objects is a difficult, time-consuming and error-prone task, because the designer must take into account application-domain requirements, which are usually broader than single-application requirements; clearly, being able to reuse existing objects (code) is a plus in any software development project.

At the start of a software project, it is unlikely that all objects of interest are identified, complete and available. Building an object-based application, creating new objects and re-using existing objects are slow and costly processes -- compared to the traditional ways of building procedural software. One of the objectives of this thesis is to describe and encourage the use of OOM, which prescribes the necessary and sufficient conditions for better-quality software production.

Software may be seen as a mass of tiny, ‘uninteresting’ detailed elements which interact over a wide range of operational conditions; no written documentation can describe (complex) software completely (what, why or how), and this is true for complex systems like SV software. When developing software, the designer commonly combines two contradictory methods, namely synthesis and analysis. For example, when writing a text chapter, he/she applies synthesis; when reviewing the text, he/she relies on analysis; all together, writing and reviewing form an iterative process which continuously combines synthesis and analysis. When writing software code, the objective is to produce a ‘text’ (the ‘source’ code) which is sufficiently readable to be analyzed. To be able to re-use components, the source codes need to be sufficiently readable and precise for object selection and re-use, whilst retaining enough flexibility for possible software extension. The application of OOM implies that the knowledge encapsulated in objects grows constantly, which enables the emergence of new systems, platforms and applications.

The selection of tangible objects is clearly a key factor for successful OO software design. Nothing definitive can be said about the ‘right way’ to choose objects: this is an acquired skill. Selecting different collections of objects would lead to different bases for developing and extending a given application. Where new objects are required, they are usually created from existing ‘correct’ ones, which prevent error-prone situations (such as unpredictable/erroneous inputs) and ensure the correct ‘trapping’ and treatment of anomalous situations. This significantly unloads the designers from basic concerns. In contrast to OOM, procedural methods are direct and focused on the processing aspects; their strength is their ability to deliver high-performance software and their weakness is in their low level of productivity and in the fact that they produce software that is ill-suited to evolution/modification. Implicit in OOM is the ability to check the logical correctness of the software, which workload is of the order of magnitude larger than the workload to improve software performance. OO software
tested under a large variety of operating conditions is expected (very likely) to work correctly in a new situation/application.

To summarize: **OOM** was selected as the appropriate methodology to develop our VS system because OOM:

- allows the developer to model real-world problems abstractly in terms of objects,
- leads the developer to think about the problem of interest in terms of the application itself, not in terms of computer concepts; it helps the developer to think without using programming concepts,
- requires the full understanding of the application requirements and promotes a design that is clean and implementation-independent,
- defers all implementation details until later stages, which preserves coding flexibility,
- builds upon a set of concepts and graphical notations which is used at all stages of the software development process, in contrast to other methodologies which use different notations at each development stage. The concepts and notations provide useful documentation during software construction,
- provides practical and effective instruments to develop, integrate, maintain and enhance software.

The fundamental concept of OOM is clearly the *object*, an entity which combines both data structure and behavior. An advantage is that the application-domain and computer-domain objects are designed and implemented using the same concepts and notations. In an **OO** approach, the software is organized as a structured discrete collection of objects which incorporate state and behavior information. That is in contrast to conventional *functional*-oriented *FO* approaches (also called *procedural* or *structural* approaches) where state and behavior are uncoupled or at best loosely-coupled. The **OO** approach places greater emphasis on data structures and lesser emphasis on procedural structures than the **FO** approach. The term *object-oriented* (OO) is an adjective employed to qualify all phases of the software development cycle, as in ‘**OO analysis**’, ‘**OO design**’ and ‘**OO implementation**’ [101]. In all development phases, we will be specifying the problem/solution domain in terms of **OO** concepts such as:

- Objects and Classes,
- Messages and Methods,
- Inheritance,
- Polymorphism.

In the **OO** approach, all entities of the problem/solution domain are represented as objects. Although the **OO** approach is based on a few simple concepts, applying them is not seen as being simple, probably because most developers have inherited a procedural (or functional) way of thinking, a natural by-product of the algorithmic logic applied in all engineering disciplines.

A short overview of the object-oriented concepts is given in the following sections.

### 3.2.1 Objects and Classes

An *object* is an entity possessing a *state* and *behavior*, see Figure 175, which describe its nature and functionality [100, 102]. Objects can have analogies in real-world things (e.g. house, ship, and aircraft) or in mathematical models (e.g. number, point, surface). When similar objects are grouped together, the Abstract Data Type (*ADT*) can be defined, to encapsulate a set of common methods and states, accessed and modified only through a message passing mechanism. In addition, the hierarchical structure of the *ADT* model allows for extensibility.

The *Object Identity* means that *ADT* are implemented into discrete and distinguishable *objects*, for example, a surface of a mesh or a node of a cell. Each such object instance has its own intrinsic identity. Thus, two objects are distinct even if all their attributes values are identical. In real world objects simply physically exist, while in software, they exist if they can be uniquely accessed and referenced: by an address, array index or unique value of an attribute.
Objects that are inside an application are identifiable, can refer to, or contain other objects. Object identity concept is implemented with data types and pointers in OO programming languages, foreign keys in data bases management systems, and file names in operating systems. The object identity is a unique way to access an object by:

1. Memory reference / address
2. User-specified name
3. Identifier keys in the collection.

1. Memory reference / address

The Memory Address of the object is an external application mechanism to relate to an object, and therefore it is application dependent. Identity has to provide a unique identification of the object and, therefore, must be an independent feature internal to an object. Without object identity it is impossible to assign self-contained objects to the class attributes or instance variables, moreover to let the same object be part of multiple objects. An object identity makes possible to distinguish one object from another.

2. Object naming

The objects are identified inside the application by names. The problem rises when a specific object has more names, thus it becomes more complicated to find out if a name identify the same object.

3. Identifier keys in the collection.

An object can be identified by a unique identity key name. For example, to access the instance of the color contour representation regarding the scalar surface one can identify it with the name “contour shaded”. The dictionary class that is used to collect all the scalar surface representations supports the functionality to extract the object associated with the “contour shaded” label. This identity key name is uniquely associated with each object inside the dictionary. The variability of the key name is not allowed because that can become a serious problem when extending or replacing names in a consistent manner by giving the possibility to access the objects in more than one way. The distinction of the objects content versus the object identity is fundamental in object oriented programming, for example, when performing the following basic tests:

- The identity test checks weather two objects are the same ones.
- The equality test checks weather the content of the two examined objects is equal.

The state of the object is represented by a set of values assigned to its attributes, also called instance variables. For example, the point can have coordinate values (5, 7). These values represent the state of the point objects. The behavior of the object is represented by the set of methods (procedures, operations, functions), which operate on its state. For example, the point can be moved. The method "move" represents the point behavior and affects the point state (coordinate values).
The classification means that objects with the same data structure (attribute, state variables) and behavior (messages, operations) are grouped into classes. A class is an abstraction that describes relevant properties and ignores the rest. Each object is an instance of its class. Each instance has its own attribute value but shares the same attribute names and operations with other class instances. The same classes design can be used without a change in notation, through the complete software development lifecycle; although at different development phases the encompassed details can vary. A class is a group or classification of objects that share the same state and behavior. The class has to provide the abstract part of knowledge, with relatively stable meaning in various contexts, tangible to the end users (e.g. CFD engineers). The class should be a clean implementation of ADT, defined through the processes of data abstraction. Data abstraction is the process of specification and implementation of ADT, see Figure 176. The specification defines what the ADT is able to do, which is in the contrast with the implementation defining how the detailed implementation is done.

Data Abstraction consists on focusing on the essential, inherent aspects of an object, before deciding on how to implement it, in order to avoid premature commitments to details. The abstraction process highlights the reasons of the object existence, by determining what is important and what not. ADT attributes exist only through the provided sets of messages offered by the ADT interface mechanism, which clearly separated the designer and the user concern of the implemented ADT.

The term object is often used to identify a class instance. An object is an instance of one and only one class. The objects of the same class share the same behavior. Data abstraction permits each class to be designed, implemented, understood and modified independently, localizing the effect of any change made. Each object is:

- is the instance of a class
- has a state, made up of the values to its instance variables.

The state is captured in the instance variables, while the behavior is captured in the methods. They can be accessed through messages directed to the object instances, and on which the object instances react. This is assured through the object identity, which is established when the object is created. The state of the object can change during the application session, but the identity remains unchanged throughout its life time. Identity is supported in C++ with pointers, as unique identifiers associated with each and every object. In C++ language syntax private and public keywords are used to separate the interface from the implementation parts of the class declaration. The interface is the part which a class user can access, while the implementation is the part which implements the interface functionality. Normally a class declaration is describing the class interface. In C++ there is a possibility to deny access to some parts of the declaration and, at the same time, to speed up the execution of some simple and frequently used functionality can be achieved by the in-lining mechanism.

### 3.2.2 Messages and Methods

An ADT can be seen as an object instance, whose state is only accessible through the class interface, see Figure 175. The class interface consists of the list of messages, that the object is capable to process. A message represents a condensed notation for invoking the appropriate method. For example, an integer data type in
FORTRAN is an ADT as for example, to modify its integer value, an assignment operator (message) has to be invoked, (e.g. I=5).

The message/method separation is present in ADT specification and implementation, see Figure 176. The message syntax specifies the rules how to invoke the class methods. The message semantics specifies the actions, which simulate the class behavior. The detailed definition of the message semantics is essential for the method implementation of the class (ADT) [100]. The application of a message to an object is called sending the message. The object class must find an appropriate method to handle the message. The message passing mechanism allows method invocations, consisting of the object identification followed by the message. The message can include any other parameter needed in method execution (e.g. aircraft-fly, point-move(x,y)), therefore the message passing mechanism provides a consistent way of communication between objects, see Figure 177.

Encapsulation or Information Hiding consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the objects, which are hidden from other objects. The implementation of an object can be modified without the need to modify the application that uses it. Thus, the encapsulation restricts the propagation of the side effects of small modifications. OOPL makes encapsulation more powerful and cleaner than conventional languages that separate data structure and behavior.

Information hiding, or encapsulation, is the principle of hiding internal data representation and details of implementation of ADT, allowing the access through a predefined interface. The interface, represented by a limited number of messages, reduces the interdependences between objects, as each one can only access other objects through their interface (messages). The interface is ensuring that some of the class attributes and methods can not be corrupted from outside. An object has three types of properties:

- a set of messages, interface, to which the object can respond,
- a set of attributes, instance variables, which only the object can access,
- a set of methods, which are invoked when message is send to the object.

![Figure 177: Point object](image)

### 3.2.3 Inheritance

Inheritance is the sharing of attributes and operations among classes based on hierarchical relationship. Each subclass incorporates or inherits the properties of its subclass and adds its own unique properties. The ability to factor out common properties of several classes into a common super-class can greatly reduce repetition within designs and programs and is one of the main advantages of an OO system. The sharing of code using inheritance is one of the main advantages of OOPL. In the procedural approach we would have two separated hierarchies: one for data structure and another for procedures structure. In the OO approach we have one unified hierarchy.
More important, then eliminating the code redundancy is the conceptual clarity in recognition that different methods are related to the same data structure, what largely reduce the number of cases which need to be specified and implemented.

Inheritance is a partial ordering of classes when the relationship of inclusions is applied among some of their properties. Ordering is usually achieved hierarchically from generic abstract super-classes at the root to subclasses of greater specialization and tangibility, see Figure 178.

A new subclass is constructed from an existing one, conceptually related super class, by inheritance, specifying only the difference between them. The inheritance can extend or restrict the features of the super class in order to create the specialized subclass. Inheritance therefore represents the realization of the generalization/specialization principle used to create new subclasses incrementally from existing less specialized super classes.

A class can have multiple number of subclasses and super classes. If the class is having only one super class we are speaking about single inheritance, see Figure 178a. The more complex inheritance is the multiple inheritance, see Figure 178b, when a class is allowed to have more super classes. Two different approaches to apply inheritance are based on ADT decomposition, see Figure 176:

- specification inheritance (dynamic binding) is used as a mechanism of sharing messages, common interface (syntax and semantics specification) between conceptually related classes when implementation is not comprehended,
- implementation inheritance (static binding) is used as a mechanism of sharing state, attributes (data representation) and methods (algorithms) implementation of objects.

An inheriting object typically acquires the implementation of its super class and can specialize by adding the appropriate functionality. Specification inheritance is more restrictive then implementation inheritance and results in conceptually tight hierarchies. These forms of inheritance have contradictory objectives in many ways, and consequently it is dangerous to attempt their simultaneous use.

C++ makes it possible to construct systems using both inheritance mechanisms. To efficiently use the hierarchy, mechanism programmers need to be able to work effectively along the top layers of the hierarchy without going into derived classes. This process enforces the selection of the appropriated class operation at run-time, which is in contradiction with data type checks and performance. OOP enables that the run-time support is activated where the compile-time support is not possible.

Figure 178: Single and multiple inheritance
The run-time identification together with inheritance provides a form of polymorphism that gives to the designers and programmers a flexible construction methodology to create software components, which are reflecting the Object Oriented Analysis and Design concept with the straightforward counterparts in OOP with C++ as follows:

- Inheritance mechanism allowing multiple base classes
- Calling mechanism through virtual functions invocation
- Virtual base classes enabling: Encapsulation, Type Checking and Data hiding.

Good software design is the result of well trained and insightful thought process of its architects, and inheritance is one of the well adapted mechanisms to promote software reuse.

### 3.2.4 Polymorphism

Another, equally important OO concept is *Polymorphism*, which is much less understood and appreciated than *Inheritance*. The term *Polymorphism* means to have many forms, and in software construction *Polymorphism* means that the same *message* can be sent to different class instances. As mentioned previously, a specific implementation of a message is a *method*. If a message is polymorphic, it may have more than one method implementing it. An *operation* is an action or a transformation that an object performs or it is subjected to.

Polymorphism is the ability to define different kind of objects (classes), which support a common interface (messages). Thus, objects with quite different behavior may expose the same interface (e.g. aircraft-move, point-move). There are two types of polymorphism in OO approach, see Figure 179:

- **polymorphic object**, where receivers of the message can be objects from different classes,
- **polymorphic message**, where the same message can be invoked with different number and/or types of arguments.

**Example for polymorphic object**: different classes as Aircraft and Point have the same message *move* inherited from the super class Object. Both classes Point and Aircraft redefine the method *move*, which is executed when the message *move* is sent to them. Here the specification inheritance (dynamic binding) mechanism is related to polymorphism. Polymorphism is essential for the implementation of a loosely coupled collection of objects, which classes are not known until they are identified by the program at run-time. Thus, the message *move* can be applied to a collection of objects without knowing if the object is Point or a whole Aircraft.

**Example for polymorphic message**: class Point can have two messages. First, *move(x)*, is a one argument message and second, *move(x,y)*, is a two argument message. In the first case the point is moved just in x direction, while in the second case the point is moved in x-y direction. Thus, it is obvious, that the application of the same message has two different behaviors.

![Figure 179: Polymorphism](image)
3.3 Object Oriented Programming in C++

Any OOP language must provide the data abstraction, inheritance and polymorphism mechanisms required by the OOP paradigm [36, 103, 104]. Each of these OO concepts may be used independently, but when applied simultaneously they nicely complement each other. When creating an object, one aims not only at fulfilling the requirements of an application, but at establishing the complete functional set that justifies the existence of the object independently of the application. An OOP language like C++ enables the developer to implement a class by specifying the interface and to encapsulate the algorithmic knowledge within its class methods.

Any implementation of an interactive visualization system comprise of the following elements:
- A set of abstract data types that represent information corresponding to certain real or abstract constructs/concepts;
- A set of operations on these data types;
- The C++ implementation that allows interaction of these data types.

Scientific VS has to implement general-geometric objects such as curves, surfaces and bodies, organized within geometrical category, which groups together one of the key data-types. The geometrical category provides operations that create new instances, which are queried, modified and processed by these classes.

Using C++ as a programming language yields the following benefits:
- High efficiency: C++ is much more efficient than other OOPL, for example Smalltalk.
- Industry-standard, production code is performing, as codes written in C or FORTRAN.
- C++ code is easier to maintain.

Using C++ imposes a discipline on the programmer who must specify his/her implementation as a set of classes. The programmer is encouraged to find ‘clean’ ways to solve problems when using C++. Also, C++ code organization simplifies the work of the programmer who has to maintain or upgrade the code.

In itself, the possibility of re-using CFView classes encapsulating various functionality sets was a sufficient reason to move to C++. In CFView, every significant element of the user’s interface part is mapped to an object, which implementation is driven by the GUI object architecture based on Model-View-Controller design, as explained later on in section 3.6. These objects are spread throughout the CFView code and they do simplify many implementation problems. A ‘Category’ is a collection of classes that capture abstractions for a specific layer, because it is important to cultivate these key abstractions, as they represent the interfacing mechanism between classes, why its specific method implementation might change more frequently. Objects are natural and ‘comfortable’ to use even when other benefits are not obvious.

The inheritance property suggests that it might be possible to turn each specific class into a hierarchy in which the general abstraction is captured in the base class. It is commonly observed that building good inheritance hierarchies requires some forethought, while retrofitting of a generalization can be painful. The application framework is a set of classes that provide basic components for the application assembling.

When a modification has to be done to C++ code, most programmers tend to stop and think. This may mean that the change will take a little longer than expected with other programming languages but a main benefit is the resulting code clarity, which is retained and rewards a programmer, especially the next time he/her has to look at the affected code.

The quality of software architecture is preserved in the abstractions captured in the C++ language classes.

The Naming convention includes:
1. class naming,
2. member function,
3. order of arguments,
4. abbreviations of common words.
The objectives of the naming convention are:
1. to eliminate name conflict during linking,
2. to provide quick identification of class and class member, (data or function),
3. to provide syntactic cues to distinguish between functions,
4. to provide a way for deriving the function name from a desired functionality.

The naming is based on three concepts:
1. an upper case letter for class names,
2. a lower case letter for class data members, and
3. a lower letter for class member function:

The name of a member function might involve three parts:
1) <result object> <name>
2) <object involved> <name>
3) <functionality descriptor> <verb>

The second part <object involved> identifies an object, which is manipulated and sometimes returned, for example the field Find member function represents a use of such naming convention. The object name is omitted in the member functions, as the function operates on an instance itself. The order of arguments in the member functions calls for input arguments to appear before output arguments. The <functionality descriptor> describes the operators like Find, Insert and Make. As earlier stated, these rules are used to facilitate the reuse of the designed class.

OO programming did not begin with the C++ language: it started in the 1960’s with the development of SIMULA, a language developed at the Norwegian Computer Center for the purpose of simulating real-world processes. SIMULA pioneered the concepts of classes, objects and abstract data-types. This was followed by LISP/Flavors and SMALLTALK in the 1970’s, and several other object-oriented implementations. However, these were languages mainly used in the research environments. It was not until the late 80’s that OOP begun to gain momentum and C++ started to be recognized as the powerful language that is still today.

The ancestor of OOP languages, SIMULA 67, was conceived for developing simulation software. The first complete, stand-alone object-oriented development systems were built in the early ‘70s by XEROX at its Palo Alto Research Center. The main aim of Xerox’s first research on SMALLTALK systems was to improve the communication between human beings and computers in the Dynabook project [31].

Artificial intelligence research -- especially expert systems -- had also a strong influence on object-orientation. The developers of the time were guided by concepts like those of Marvin Minsky, first described in his “frame”-paper and later summarized in his “The Society of Mind” [105]. Minsky and other authors explained the basic mechanisms of human problem-solving using frames, a concept well-suited to computer implementation. This explains the similarities between classes in systems like SMALLTALK [31] and LOOPS [106], and units in KEE [107].

Programming methodologies are used to structure the implementation process. The current state-of-the-art suggests that OOP results in greater benefits in implementation than other methodologies (e.g. structured programming), namely:
- simplification of the programming process,
- increase in software productivity,
- delivery of software code of higher quality.

OOP is a set of techniques for designing applications that are reusable, extensible, scalable and portable across different computers platforms. The purpose of designing re-usable code is to lower the time (and cost) of producing software. Experience shows that, no matter how well specified the application requirements are, some
requirements are bound to change and this results in the need for further development. The requirements change frequently for the user interface and for the system environment/platform, while they tend to remain quite stable for the underlying algorithms. Designing extensible code facilitates the changes that are necessary to meet modified requirements. Designing reusable code ensures that code used in today’s applications can be applied in tomorrow’s applications with minimum modifications.

To implement an abstract solution well-specified in terms of OOP concepts and constructs, we need a programming language with rich semantics, i.e. which can directly express the solution in a simple syntax. OOP is a methodology for writing and packaging software using an OO programming language. C++ is an extension of the C language which includes OOP support features; it was developed at AT&T in the early 1980’s [30, 108, 109]. OOP concepts have been well integrated into C++, and a very little amount of new syntax has to be learned when moving from C to C++.

The fundamental OOP features supported by C++ are:

- data abstraction and encapsulation,
- message passing,
- inheritance.

In software engineering, and particularly in OOP, it is important to manage the complexity of implementation; OOP requires a software-development environment which comprises the following components:

- text editor,
- class, data and file browser,
- compiler and linker,
- debugger,
- class libraries.

The programmer’s creativity is best enabled when the development environment is accessible through an integrated user interface giving total control over all aspects of the implementation process. Such an environment also constrains the developer’s moves and code must be produced and changed in a disciplined, consistent way.

OOP requires browsing techniques in order to identify the classes and their interactions. Software productivity and quality are significantly improved in a development environment that can display class relationship information and is able to present just enough necessary and sufficient data for the developer to manage the complexity of the code and of the coding process. The complexity is usually is layered through class categories and hidden through the inheritance mechanism. The real benefit to the programmer is that, code changes are bounded a class when single functional change is done. To locate the code influenced by a change, the class including the functionality is the basis for the scope of the change. If the class is a derived one, the tool must be able to show all the functions that it has access to, not just the ones that it defines. An integrated development environment must help to direct the developer on what he/she wants to do rather than on how to do it.

A class library is composed of header files and a library. The header files with extension “.h” can be located in the include directory. They contain all the information needed by the programmer for using the class. The files that contain the implementation of class methods (member functions) are archived in the library in compiled form. The library can be located in the “lib” directory with the “.a” extension.

3.4 Example of Object-Oriented Software Development

It is obvious that all these concepts and techniques can not be grasped, mastered and successfully applied without practice. The developer’s experience is very important in considering all the factors that influence the implementation process and in balancing them to achieve optimum trade-off. The example below -- visualization of a streamline -- illustrates how OOP is applied to software requirements analysis and development.
3.4.1 Requirements, Analysis and Design Example

Requirements analysis typically begins with a ‘narrative’ description of the problem domain. We consider here the problem of how to visualize a streamline. This example is simplified to best explain the different modeling techniques used to tackle the problem.

The problem is to visualize the streamline that passes through the point that the user has picked on the screen with a mouse click. The streamline can be associated to the pressure distribution. The user can ask for the streamline to be displayed; he/she can ask for the streamline to be colored to reflect the pressure values along the streamline.

Narrative problem descriptions are the source of many difficulties in OO analysis because the identification of the objects and their behaviors are not explicit; several standard software modeling tools must be applied to correctly identify and describe the objects of interest, including:

**GRAPHICAL MODELING TOOLS:**
- *Data flow diagram* (DFD), for functional modeling (see Figure 180),
- *Entity-relationships diagram* (ERD), for data modeling (see Figure 181),
- *State-transition diagram* (STD), for the modeling of interactive system behavior (see Figure 182);

**TEXTUAL MODELING TOOLS:**
- *Data dictionary*, for data specification.
- *Process specification*.

These modeling tools identify different characteristics of the same object; these must uniquely represent the object and prove the necessity of the object’s existence. The DFD is a graphical representation of the functional decomposition of the system in terms of processes and data; it consists of (see Figure 180):
- *processes*, shown as "bubbles", (e.g. calculate streamline),
- *data flows*, shown as curved lines that interconnect the processes, (e.g. streamline geometry),
- *data stores*, shown as parallel lines and which exists as files or databases, (e.g. velocity field),
- *terminators*, shown as rectangular boxes. They show the external devices with which the system communicates (e.g. screen, mouse).

![Figure 180: DFD of the streamline example](image-url)
The data flow diagram identifies the major functional components but does not provide any details on these components. The textual modeling tools -- the data dictionary and the process specification -- respectively provide details on the data structures and data transformations. For example, the ‘Point’ data dictionary and the ‘Process 3’ specification are as follows:

**Point:**
point = coordinate x + coordinate y
coordinate x = real number
coordinate y = real number
real number = [-10 -6 - +106]

**Process 3:**
1. Find the cell of the first streamline point.
2. Interpolate the pressure.
3. Search for the next streamline point inside the cell.
4. If point found, interpolate the pressure, else continue the search through neighbor cells.
5. repeat actions 3-4 for all streamline points.

The DFD is a useful tool for modeling the functions but it says little or nothing about data relationships. The data-stores and the terminators in the DFD show the existence of one or more groups of data. One needs to know in detail what data is contained in the data-stores and terminators, and what relationships exist between them. The ERD tool is used to model these aspects of the system and is well-suited to perform an OO analysis of the data and of their relationships.

An ERD comprises as main components (see Figure 181):

- **entities**, shown as rectangular boxes, each representing one or more attributes, (e.g. Point, Curve),
- **relationships**, shown as diamond-shaped boxes, each representing a set of connections, or associations, (e.g. HAS, Streamline HAS Nodes),
- **attributes**, shown as ellipses. They cannot contain entities, (e.g. Point coordinates).

![Figure 181: The partial ERD of the streamline example.](image)

Each relationship is expressed as a *multiplicity statement*, which can be one of three possibilities:

- one-to-one - 1<>1,
- one-to-many - 1<>M,
- many-to-many - M<>M.

In Figure 181, one Streamline has many Nodes. The relationship is ‘HAS’, and the multiplicity of the relationship is one-to-many, 1<>M.

A third aspect of the system that needs to be described is its time-dependent (real-time, interactive) behavior. This behavior can be modeled by a state-transition diagram using sequences which show the order in which data will be accessed and functions performed. To model the streamline example with a STD (see Figure 182), we need to add the *conditions* that cause changes of state and the *actions* that the system will take in response to the changes of state (e.g. click of mouse button -> get coordinates). A condition is an event in the external environment that the system can detect (e.g. mouse movement, clicking of mouse button).
For example in Figure 182, the system is waiting for point coordinates. Pressing the mouse button will cause the system to change from the state "waiting for coordinates" to the state "waiting for choice".

The top-down decomposition, applied to the streamline example, shows the direct correspondences between data (functions) and object states (behaviors). These correspondences (data - object states) and (functions - object behaviors) -- help us in classifying and understanding the system components and in finding/defining the objects.

An OO analysis that uses the set of the presented modeling tools permits to identify and capture objects that are systematically defined as part of a desired solution and thus they can be checked in advance (before implementation occurs); which is somehow in contrast to a subjective and possibly inconsistent "users view" of a problem to be solved.

The OO analysis has led to defining a set of objects and relationships that correctly depicts the real world and meets the user requirements. These objects must perform various (types of) operations and must interact; conventions must be defined to enable such interactions by means of messages.

OO analysis consists of these basic steps:

- identify the objects,
- identify the attributes and the messages of the objects,
- identify the relationships between the objects.

Generally, the objects are identified as nouns in the process specification, and also as data in the data dictionary (e.g. Point). The most appropriate way to find the objects is from the ERD, because of the (usual) one-to-one correspondences between entities and objects. In this case, entities have also to be present in the process specification and data dictionary. In the ERD of the streamline example, we can extract the objects: ‘Point’, ‘Quantity’, ‘Node’ and ‘Streamline’.

The object behaviors can be identified in the DFD. For example, the streamline can be calculated from the velocity field. Hence, the ‘Velocity Field’ must have a method for calculating the streamline when it receives the ‘Streamline’ message with the ‘Point’ argument. The identification of the object behavior can be found in the STD. For example, the ‘Mouse’ object must return the ‘Point’ object when the mouse button is pressed.

Three basic relationships may be identified between objects:

- ‘has’ - indicates that one object contains another one, (e.g. ‘Curve’ has ‘Points’)
- ‘is a kind of’ - indicates that one object is a specialization of another one, (e.g. ‘Node’ is a kind of ‘Point’),
- ‘uses’ - one object interacts with another (e.g. ‘Velocity Field’ uses ‘Point’ to calculate ‘Streamline’).
For the streamline example, a preliminary class specification would be:

<table>
<thead>
<tr>
<th>CLASS NAME</th>
<th>ATTRIBUTES</th>
<th>MESSAGE</th>
<th>MESSAGE RETURN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse</td>
<td>point</td>
<td>buttonDownEvent()</td>
<td>Point</td>
</tr>
<tr>
<td>VelocityField</td>
<td>velocities, cells</td>
<td>streamline(Point)</td>
<td>Streamline</td>
</tr>
<tr>
<td>Streamline</td>
<td>points</td>
<td>geometryStructure()</td>
<td>Structure</td>
</tr>
<tr>
<td>PressureField</td>
<td>nodes, cells</td>
<td>streamlineStructure()</td>
<td>Structure</td>
</tr>
<tr>
<td>Screen</td>
<td>-</td>
<td>display (Structure)</td>
<td></td>
</tr>
</tbody>
</table>

The essential part of the OO analysis phase is the documentation, which include the specification of the objects with their attributes and messages. The methods have to be described in sufficient detail to ensure that the application requirements are complete and consistent. These classes represent the problem space, the first layer of abstractions.

Design can start as soon as we have the analysis of the problem. It is important to ensure that the analysis specification can realistically be implemented with the available software development tools. OO design is a bottom-up design: the lower-level classes can be designed before the high-level classes. To take advantage of reusability, it is natural to separate the classes identified during analysis in two groups:

- the classes to be developed - new components,
- the classes to be reused - existing components (class libraries).

It is difficult to design an entire workable system without prototyping some parts of it and exploring alternative solutions based on existing class libraries. The class libraries relate to distinct computer-system areas -- e.g. graphics, user interface, mathematical, etc. -- and constitute frameworks that could be integrated to form a workable solution. If we assume that the class libraries ‘Continuum’, ‘InterViews’ and ’PHIGS’ are available, the classes that must be designed are ‘Streamline’, ‘Velocity Field’ and ‘Pressure Field’. OO design relies on two graphical modeling tools for identifying the new classes, namely:

- the ‘class hierarchy’ diagram,
- the ‘class attributes’ diagram.

The class hierarchy diagram (see Figure 183) of a class library provides an immediate means of positioning a new class within the library. For example, in the class library ‘Continuum’, we have the class ‘Scalar Curve’ which is very well-suited for Streamline implementation; also, the classes ‘Pressure Field’ and ‘Velocity Field’ are instances of the ‘Scalar Field’ and ‘Vector Field’. Hence, we can start with the implementation phase and prototype the application.

Because the classes of the streamline example are existing in the above class hierarchy diagram, the example of the class attribute diagram (see Figure 184) is given for the involved Continuum classes.
The design phase can stop when all the objects that have been defined are ‘simple’ enough, that is to say that they do not require any further decomposition. Thus, we have composed all the required objects from the existing classes and the implementation can follow.

### 3.4.2 Data Abstraction and Encapsulation

The key concept in C++, as required by OOP paradigm, is the concept of ‘class’. The class implements the state and behavior of an abstract data type (ADT) [110]. Data abstraction is supported through the class mechanism. Data abstraction refers to the separation of the class interface from the class implementation. Abstracting (separating) the class interface from its implementation also means that the class implementation can change without affecting the client code that uses it. Data abstraction is normally used for the definition of primitive data types (e.g. INTEGER, REAL in FORTRAN or int, float in C). For example, when using floating point arithmetic in C, a programmer does not need to know HOW floating point numbers are represented internally, nor HOW floating-point arithmetic is performed: he/she merely needs to use the data type:

```c
float a=1, b=2;
a+b;
sqrt(a);
```

With the C++ syntax, a class is specified as a set of data (variables) together with a set of functions. The data describes the object state and the functions describe its behavior. A class is a user-defined data type, giving the user the capability to define new data types. For example, the following is the definition of the ‘Point’ class:

```c
class Point
{
private:  // REPRESENTATION - object state
    float xx;  // x coordinate
    float yy;  // y coordinate
    float zz;  // z coordinate

public:  // INTERFACE - messages
    void move // increase coordinates
        (float dx,  // x direction
         float dy,  // y direction
         float dz); // z direction
    float x();  // return x coordinate
    void x // modify x coordinate
        (float v) // new x coordinate
        {xx=v;};
};
```

In C++, the class definition mechanism allows to declare variables called data members (e.g. float xx) and functions called member functions (e.g. move). The class definition is a little bit different from the ADT decomposition; because it puts together the class representation and interface (see Figure 176).

The access to actual data is controlled by admitting only the use of the messages associated with the object; this prevents indiscriminate and unstructured changes to the state of the data. The data members are said to be encapsulated since the only way to get at them is by use of the member functions (invoke messages) associated with the methods implementation (e.g. move).

```c
void Point::move  // move method implementation
    (float dx, float dy, float dz)
    {xx+=dx; yy+=dy; zz+=dz;};
```
The message passing mechanism forms a sort of software shell around the class. The C++ keywords private, protected and public support the technique of encapsulation by allowing the programmer to control access to the class members. In the class definition, member data and functions can be specified as public, protected or private:

- **public members**, typically functions, define the class interface,
- **protected members**, typically functions, define the class interface when inheritance is applied,
- **private members**, typically data types, can only be accessed inside member functions implementation (methods). They reflect the state of the object.

The distinction between ‘public’ and ‘private’ members separates the implementation of classes from their use. As long as a class interface remains unchanged, the implementation of a class may be modified without affecting the client code that uses it. The client programmer, user of the class is prohibited from accessing the private part. By default, class members are private to that class. If we have to change point coordinates, we have to use appropriate member functions, for example x to update the x coordinate. There are three ways to identify the object in C++:

```
Point a; a.x(10); // by name
Point& b=a; b.x(10); // by reference
Point* c=&b; c->x(10); // by pointer
```

In this example, the three points a, b and c represent the same object. In the class Point definition, the implementation of x member function is also present. By default such function implementation is expanded as inline function. Thus, there is no function call overhead associated with its use.

The following definition extends the ‘Point’ class with two new member functions read and write allowing the input and output of the Point instance coordinates:

```cpp
class Point
{
private: // REPRESENTATION - object state
    float xx; // x coordinate
    float yy; // y coordinate
    float zz; // z coordinate
public: // INTERFACE - messages
    void move // increase coordinates
        (float dx, // x direction
         float dy, // y direction
         float dz); // z direction
    float x(); // return x coordinate
    void x // modify x coordinate
        (float v) // new x coordinate
        {xx=v;};
    ostream& write // output
        (ostream&);
    istream& read // input
        (istream&)

    ...;
};
```

In the definition of the ‘Point’ class, we have introduced 2 new classes istream and ostream which are standard classes in the C++ I/O stream library. Instances of the output stream cout and input stream cin are defined by default. We can use their << and >> operators without caring about their implementation.

The implementation of the read and write member functions is as follows:

```cpp
    istream& Point::read(istream& s)
    {return s>>xx>>yy>>zz;};
    ostream& Point::write(ostream& s)
    {return s<<"(*<<xx<<","<<yy<<","<<zz<<")")";};
```

In the body of the member function implementation, any undeclared variables or functions are taken to be member variables or functions of the class (e.g. xx, yy, zz).
The default implementations of the << and >> operators are illegal too for the ‘Point’ class, thus they have to be implemented in terms of read and write operators.

```cpp
    ostream& operator<<(ostream& s, Point & p)
    {return p.write(s);};
    istream& operator>>(istream& s, Point & p)
    {return p.read(s);};
```

In order to improve performance in the use of such functions, they may be defined inline.

In the following example member functions are called using the normal member selection operator (e.g. -> or .). An object or pointer to an object must always be specified when one of the Point member functions is called.

```cpp
    Point a;
    Point *b=&a;
    a.read(cin);
    b->write(cout);
```

These input and output member functions may be called from anywhere within the scope of the declaration of the Point instance. The same functionality can be achieved with the new definition of the << and >> operators.

```cpp
    Point a;
    Point *b=&a;
    cin>>a;
    cout<<*b;
```

Operator functions may be members of a class. For example, a += operator could be declared as a member function in the class definition by:

```cpp
    Point& operator += (Point p);
```

and the implementation of the operator as:

```cpp
    Point& Point::operator+=(Point p)
    {xx+=p.x();  yy+=p.y();  zz+=p.z();  return *this;};
```

Often, member functions require state information to perform their operation. Since every instance may be in a different state, the state information is stored in the data members. Thus, the member functions provide access to the data members, whilst these support the functionalities of the member functions.

Some of the most important member functions are constructors and destructors (see class ‘Curve’). One can imagine that a class prescribes that the layout of allocated memory should be a set of contiguous data fields. Constructors and only one destructor per class are member functions which allocate and de-allocate memory during the lifetime of the object.

Memory allocation during initialization can be:

- automatic
- dynamic
- static

In the following example, the three types of initialization are shown:

```cpp
    main(){
        ...  // start of block
        Point a;  // automatic
        Point *b=new Point;  // dynamic
        static Point c;  // static
        ...
    };  // end of block
    ...
```

- automatic objects are allocated on the program stack. Their lifetime is the execution time of the smallest block,
• **dynamic objects** are allocated in free storage. The programmer explicitly controls the lifetime of dynamic objects by applying the `new` and `deleted` operators.
• **static objects** are allocated statically: storage is reserved for them during program execution.

The dynamic control of memory allocation allows the programmer to write a parameterized program. Such a program, when running, will allocate the amount of memory just required in free storage depending on the data being processed. That is in contrast with FORTRAN code where memory must be allocated in advance for handling the maximum data size (FORTRAN ALWAYS allocates the maximum specified memory).

The following example defines the class Curve representation in C++:

```cpp
class Curve{
private:  // REPRESENTATION - object state
    Point* store; // pointer to array of
    // points
    int size; // number of points
public:  // INTERFACE - messages
    Curve(int sz=10) // default number of points
        {size=sz; store=new Point[size];};//
    ~Curve() // destructor
        {delete[size]store;};
    ...
};
```

This example uses several features of C++:
• a default argument for the constructor function `Curve(int sz=10)`,
• a destructor function `~Curve()`,
• the free store operators `new` and `delete` to allocate storage.

### 3.4.3 Inheritance and Polymorphism

Inheritance is called `derivation` in C++. Classes can be arranged in a hierarchy in C++: a sub-class (of a class) is called a derived class, while the super-class is called a base class. Derived classes may in turn have classes derived from them. Hence a tree-structured set of classes can be built, known as a single inheritance.

For example, the ‘ScalarCurve’ class can be derived from the ‘Curve’ class and the ‘Node’ class from the ‘Point’ class. Conceptually, both derivations are related to the specific quantity defined for the new derived classes. The ‘Node’ derived class extends the definition of the ‘Point’ base class, so that all the members of the ‘Point’ class are also members of the ‘Node’ class. The implementation is as follows:

```cpp
class Point {...};
class Node: Point
{
    private: float val;   // quantity value
    ...
};
```

Instances of base and derived classes are objects of different sizes because they are defined with different class members. A derived object can be converted into a base class without the use of an explicit cast, i.e. without an explicit request for conversion by the programmer. This standard conversion is applied in initialization, assignment, comparison, etc. For example, a Node object can always be used as a Point object.

```cpp
Node b;
Point& a=b;
```

Standard conversions allow the base classes to be used to implement general-purpose member functions, which may be invoked through base class references without being aware of the object’s exact derived class.
Any derived class inherits from its base class its representation (including the external interface messages) and methods implementation. The derived class can modify the methods implementation and add new members to the class definition, therefore:

- the internal representation can be extended, or
- the public interface can be extended (or restricted).

The same member function name can be used for a base class and one (or more) derived classes. These polymorphic member functions can be individually tailored for each derived class by defining a function as ‘virtual’ in the base class. Virtual functions provide a form of dynamic binding (runtime type-checking). This mechanism works together with the derivation mechanism and allows programmers to derive classes without need to modify any member function of the base class. Virtual functions allow the flexibility of dynamic binding while keeping the type checking of the member functions signatures.

```cpp
class Point {
private:  // REPRESENTATION - object state
    float xx; // x coordinate
    float yy; // y coordinate
    float zz; // z coordinate
public:  // INTERFACE - messages
...
    virtual ostream& write // output
        (ostream&);        
...
};

class Node: public Point {
private:  // REPRESENTATION - object state
    float val; // quantity value
public:  // INTERFACE - messages
...
    ostream& write // output
        (ostream&);        
...
};

main(){
    Point* store[2];   // set of objects
    store[0]=new Point(...) // Point initialization
    // output for both objects
    for(int i=0;i<2;i++) store[i]->write(cout);
...}
```

In the above example, the member function (message) `write` will invoke two different `write` implementations (methods), respectively of the Point and Node classes.

When a member function is declared `virtual` in a class, all the objects of that class are labeled with the type information as they are created. This virtual declaration adds some extra storage to the class. Any member functions may be virtual, except constructors. Operator member functions including destructors may be virtual. From the behavior point of view it is frequently useful to identify the common features provided by more than one class. Most classes need to be categorized in more than one way. The mechanism of multiple inheritance allows to extend the features of a derived class using more than one single base class. For example, the class ‘Node’ could be implemented differently if for example, the following Scalar class would exist.

```cpp
class Scalar {
private:  // REPRESENTATION - object state
    float val; // quantity value
...
};

class Node: public Point, public Scalar
{};
```
This version of the class Node is composed of the Point and Scalar class giving the features of both classes to the Node class without any additional coding.

Besides virtual member functions (polymorphic object), C++ supports function and operator overloading (polymorphic message). Several member functions and operators can coexist having the same function name or operator symbol. The compiler uses the member functions arguments types to determine which implementation of a function or operator to use.

```cpp
class Point {
    ...
    float x(); // return x coordinate
    void x(float v) // modify x coordinate
        {xx=v;};
    ...
};
```

In this example, member function `x` is type of polymorphic message.

### 3.4.4 Templates and Exception Handling

In special cases, when inheritance cannot be applied, a generic class mechanism based on class templates can be used. It supports the automatic generation of individual classes, built upon the predefined code structure defined by the templates library.

For example, the method `count`, which counts the number of elements in a collection, can be defined before specializing it for the `PointCollection` or `SurfaceCollection` classes. In fact, generic classes are meaningful only in typed languages, where each class is declared as belonging to a certain homogeneous collection. Hence one can determine whether a method is correctly invoked simply by looking at the source code (static type checking) or by checking it at execution time (dynamic type checking). Dynamic type checking can also be achieved by inheritance. Generic classes are very often used in larger systems and are considered as one of important mechanisms to eliminated/reduced performance penalties associated with the use of inheritance.

Exception handling provides a consistent method to define what happens and prescribe a suitable system response when a class client misuses an object. This is a central issue when developing interactive visualization systems since one cannot avoid (unwanted) situations of implementation/ or coding anomalies. Exceptions refer to program states corresponding to run-time errors like: hardware failures, operating system failures, input errors, system resource shortages (e.g. failure to allocate a request for memory), etc. Exceptions are control structures that provide a mechanism for handling such errors. Not unsurprisingly, OOM handles ‘exceptions’ by adding them to class methods where they naturally belong; this requires the class designer to consider all possible exception situations and systematically define the methods for error handling.

### 3.4.5 Dynamic Memory Management

The concept of shared and non-shared objects is central to memory management; this includes the use of the control and free operations. In order to safely delete a shared object, one must ensure that the delete operation will not corrupt other objects which may have been using it. When objects are created, they can be used in different contexts, and multiple references to the same object can exist within an application.

A ‘set of references’ guarantees that an object is free when it is no longer needed. In order to support the referencing mechanism of shared objects, the shared objects are separated into:

- controlled shared object, and
- free shared object.

Shared objects are typically used when sub-objects are constructed as specific parts of a larger one. The shared object ensures that the sub-objects are maintained in memory until needed. For example, the boundary surfaces
are defined as shared objects, since more than one region (cell-zone, segment) can be defined from existing boundaries (shared objects), which must be retained until all zone cells (segments), which are depending on them exist. Special types of shared objects are:

- pointers to functions
- exception objects
- error states
- library signals and long jump facilities

The applied solution to this shared objects referencing problem is implemented in the base class which manages the reference counter and through virtual destructor redirect allocation/reallocation of object memory to the derived classes. Derived classes implement appropriate memory allocation algorithms, when the related derived class instance is created or deleted.

OOM includes a few other concepts like concurrency, persistence and garbage collection. They are mentioned here for completeness:

- **Concurrency** refers to the concept of operations carried out in parallel as opposed to sequentially (computers naturally operate sequentially). Concurrent systems consist of processors that operate, communicate and synchronize so as to perform a task as efficiently and rapidly as possible. For example, computational processes can be distributed and executed on several processors (distributed environment) that run on different platforms (parallel/vector/grid clusters); hence, a Process class can be designed to encapsulate the sequence of dispatch actions that supports the execution of parallel protocols.

- **Persistence** refers the activating/deactivating mechanism which permits to automatically convert (arbitrary) objects to symbolic representations. The object representations are stored as files, and the objects can be regenerated from these files by inverse transformation. Thus, persistence provides the methods for saving and restoring objects between different sessions.

- **Garbage collection** refers to the automatic management of certain types of objects. A time comes when (previously created and used) objects are no longer needed. The system must ensure that un-needed objects are deleted ones (and once only) and can never be accessed afterwards. This is the purpose of garbage collection.
3.5 Mapping Entity-Relationship model to Object-Oriented model

The analysis of the user requirements is the first step towards a more detailed software description where the trade-off between suggested visualization system functionality and the one that will be designed, is elaborated. The practice shows that the software development is not efficient, if the analysis is not carefully considered. To facilitate the design of the OO data model ERM was applied, as applied systematically in chapter 1. ERM was found useful for the description of the logical aspects of the visualization system, especially when requirements are discussed with the end users. The important element in the analysis phase is the establishment of the Naming Convention, which has to be respected through all the software development phases. Nouns are typically used to define the names (labels) of the attribute types, while verbs are usually used to name relationship types in ERM. For the creation of the naming standard of the attribute types, the following two approaches are considered:

1. To define both verb and noun for each attribute type, describing the association, as data or process depending on the context. It is obvious that such approach introduces flexibility, but also increases the complexity of the model, as it is not uniquely defining the related attribute type.

2. To define only noun or only verb to uniquely define the attribute type. This is more restricted approach but it results in a more precise software specification.

The second choice was applied, where attribute types are uniquely defined with a verb or a noun: the entity type, associated with noun and the relationship, associated with verb, while attributes for an entity or a relationship are or nouns or verbs. With applying ERM, we grouped the relationships and attributes within defined entities in order to progress in the design of OO classes. In this modeling process entities are mapped to classes. The side effect of this process is also the creation of auxiliary classes, which were not directly identified in ERM, but are found necessary to support the envisaged software functionality. The primary goal of the analysis model is to put in evidence the end user concepts and prepare a foundation for the classes design.

Attribute Type (AT) defines a collection of identifiable associations relating objects of one type to objects of another type. For example in ERM, Boundary is AT of the Zone entity modeled as an attribute of the Zone class, called Boundary. This Boundary object has the attribute, which is associating a Boundary object with one or more Zone objects, which are bounded with this boundary.

Scenarios provide outlines of the user activities that are defining the system behavior. They provide more focused information related to the system semantics and serve as the validation elements of the software design. Scenarios provide means to elaborate the system functionality, modeled usually through collaboration of the respective classes. For example, the class View often participates in several scenarios: for rendering object on the screen, interacting with the windowing system when resizing, making icons and updating the display. The most complex scenarios cut across large parts of the software system architecture, touching most of the classes constituting the application. The scientific visualization system is an interactive application, with variety of such scenarios driven by external events. Each defined visualization tool is constructed around a specific scenario, which is found sufficiently independent, thus it can be developed independently, although in practice, it could have some semantic connections with other scenarios.

ERM defines the static data structure consisting of entities, their relationships and related constraints. This modeling technique was extensively used in Chapter 1, to define the Visualization System data model. In OOM, the data structure is defined within classes. Entities and relationships are modeled as classes and there is no need for the specification of specific entity key attributes. The entity attributes are defined also as classes, which are modeled, or as an individual class or as collections of classes. The following distinctions need to be considered before the mapping is made:
Relationships are represented explicitly in ERM, but in OOM they are designed as the class references. In ERM the relationship model expresses the semantics, cardinalities, and dependencies between entity types.

Key attributes uniquely distinguish entities in ERM, which is not the case in OOM as the class instances have their run-time identifier. However, these key attributes can be of use, if the persistence or the search on such objects has to be implemented, based on an association with an indexing or labeling scheme.

Methods are not present the ERM notation. When there are constraints that cannot be specified declaratively within a class, we model such constrains with methods in order to support their verification. The methods become an integrated part of the class design.

The steps to follow when ER model is mapped to OO model:

- Entities and relationships are mapped to classes.
- Methods are added to classes.
- Additional methods are designed to support constrains, which cannot be specified declaratively.
- Entity type is mapped one-to-one to a class. The entity type hierarchy is mapped to the OO class hierarchy. Common entities are mapped to a specific subclass applying the multiple inheritance mechanism. A composite attribute is mapped to its component attributes as a collection of attributes.
- Binary relationship is modeled as object reference within a designed class. The bi-directional reference facilitates the bi-directional navigation to support such inverse referencing. If the cardinality of a relationship is greater than one, it is appropriate to model it as a collection of objects references. If a binary relationship type has one of more attributes, then it is appropriate to design a new class to handle these attributes and keep track of objects references of the two related classes, for which the bi-directional communication is required.
- Constraints can be specified for entity and relationship types. The multiple inheritance mechanism is a possibility to enforce the constraint of overlapping or disjoint entities in an entity type hierarchy.
- Cardinality constraints in a relationship type are implemented in methods for constructing or destroying objects and methods for inserting/deleting members into/from a collection of referenced objects. If these cardinality constraints exists a dependency constraint for the relationship type, should be implemented when constructing and destroying such objects.

### 3.6 Model-View-Controller Design

The visualization system design is built around an event-driven, multi-window, iconic GUI. It separates the event management (Control layer), the appearance of the application data (the View layer) and actual modeling of the application data and processes (the Model layer). This type of framework, known as the Model-View-Controller (MVC), comes from the Smalltalk programming environment [31]. MVC is a factorization of the application’s interface such that the system architecture is decomposed in independent components that separate application model from user interface model. MVC is appropriate for designing the visualization system’s GUI which integrates the specialized interfaces of many visualization tools.

The main MVC decomposition of the VS architecture distinguishes three components:

1. **MODEL** what application does logically?
Supports functionality, which consists of the visualization algorithms and CFD data management elements. It handles the internal data and operations, which are isolated from screen display, keyboard input and mouse action.

2. **VIEW**  
*how application displays itself visually?*

Represents the graphics display presentation, responsible for the displayed graphics as the feedback to all the user interactions. It includes sub-layers with direct procedure calls to window management and graphics kernels functionality.

3. **CONTROLLER**  
*how application handles input?*

The mediating components between the CFD data model and the viewing interface. It controls the user interaction with the model, including the system limitations that constrain the system operations in particular circumstances. It provides initial creation of the interactive environment and maps the user input into the application.

![Figure 185: The MVC model with six basic relationships](image)

The MVC model is applied to the overall system architecture and to the individual visualization tools that supporting the user interaction model. In Figure 185, six MVC relationships are applied to design the visualization system, especially concerned with the user interaction modeling, as follows:

1. In the model->view relationship, the Model informs the View that its state has changed. This dependency relationship ‘tells’ the View to update itself.

2. The view->model relationship allows the View to request from the Model the data necessary for updating itself.

3. The view->controller relationship allows a View to activate a Controller that depends upon its state.

4. The controller->view relationship allows the modification of the View due to an input. It is not necessary to identify the controller through 1 and 5. This is useful for View manipulations when the Model of the view does not change (for example in the case of zooming or when rotating a scene).

5. The controller->model relationship allows the modification of the model due to an input.

6. The model->controller relationship allows a Model to activate/deactivate a Controller. For example, a controller may need to be de-activated while the Model is under computation.

This design, which entails no central control, allows coexistent objects to communicate outside of the main program execution, but any modification to a model might trigger updates of its views through requests by the associated controllers. Thus, the dependency mechanism is localized per MVC objects triad. The MVC
framework can be applied to individual classes, but also as a group of collaborating classes. The envisaged architecture organizes the software in interchangeable layers, which are perfectly in line with the applied MVC framework. Such separation allows software updates to be performed without need for re-implementing the main application structure, and allows for the easier integration of components -- for example, when creating different layouts of the GUI. This separation also permits the creation of specialized layers only applicable to specific hardware platforms.

We distinguish the following layers in the VS architecture:

1. The layer that handles the input data (and uncoupled from any graphical output), for example a display screen or printer device.
2. The layer which is tightly linked to the graphics engine, i.e., the explicit invocation of graphics kernel functionality.
3. The layer which coordinates the application model and the GUI.

These features make object-oriented systems different from traditional ones in which these layers are mixed together. The possibility of independently changing the View layer or the Controller layer makes software maintenance easier, including adaptations to different display or input devices. For example, keyboard (or mouse) input devices can be changed altered without affecting the application structure.

Experienced designers solve problems by re-using proven, successful solutions or solution patterns: they can be re-applied and need not be rediscovered. This is why the visualization system reuses the MVC solution and its mechanism of dependency, which allows a change in a model to be broadcast to all objects concerned and to be reflected in the multiple views.

The MVC pattern consists of three kinds of objects:

1. the model, which is the application object,
2. the view giving screen presentation, and
3. the controller which defines the way the user interface reacts to user input and model output.

Model and View are coupled through the subscribe/notify protocol. A view reflects the appearance of the model. Whenever model data change, the model notifies its dependent views; in response, each view must update itself by accessing the modified values and updating its appearance on the screen. This approach enables the creation of multiple views of a model. The model contains data that can describe several representations. Views are nested and MVC defines the way a view responds to a user command given via the associated input device. The viewing control panel in CFView is designed with a set of buttons modeled as a complex view, which contains the views of the related buttons. The views are contained in, and managed by the Window.

The interaction mechanism is encapsulated in a Controller object. The Controller class hierarchy supports the design of a new Controller as a variant of an existing one. A View encapsulates the interaction mechanism through the interface of the controller subclass. The implementation can be modified by replacing the controller instance with another one. It is possible to change a View and a Controller at run-time. The sharing of a mouse, keyboard or monitor by several visualization tools demands communication and cooperation. Controllers must cooperate to ensure that the proper controller is selected to interpret an event generated via an interaction component, which contains (is attached) to the user-mouse-cursor interaction.

The Model has a communication link to the View because the latter depends upon the Model’s state. Each Model has its own set of dependent Views and notifies them upon a change. In principle, Views can recreate themselves from the Model. Each View registers itself as a dependent component of the Model, sets its controller and sets its View instance variables. When a View is destroyed, it removes itself from the model, controller and sub-views. Views are designed to be nested. The top View is the root of its sub-Views. Inside a top-View, are the sub-Views
and their associated Controllers. A single control-thread is maintained by the cooperation of the Controllers attached to the various Views. Control has to result in only one Controller selected to interpret the user input: the aim is to identify the one that contains the cursor. The identification of the cursor position is computed through the traversal of the associated Views.

The Model is an instance of a class MForm and consists of application data. A change to a Model is a situation which requires a modification of the View, an instance of VForm. MForm messages enable VForm to update by querying the Model data. The VForm::update is selective as determined by several parameters of the model. The VForm::display controls the graphical appearance of the Model. It is important to note that no element of the interface (view/controller) is coded in the application (model). If the interface is modified, the model is not influenced by this change. The MVC framework is established using the 3 basic classes MForm, VForm, CForm, see Figure 186 and follows the example that describes the three specialized classes for the treatment of the Surface model.

Figure 186: MVC framework for Surface manipulation

MForm is completely application-dependent and consists of key application data. In our surface example this is the surface appearance. VForm is the class which contains the attributes that set the appearance of the surface on the screen, for example the wireframe thickness parameter and its color. CForm is the class which controls the input devices -- such as dialog boxes used for selecting the color. MForm does not define the user interface directly; hence any message related to View, display of surface and to controller, modifying the surface appearance is avoided inside it. The trigger mechanism designed through the dependency mechanism informs the
model’s dependents of its change. VForm and CForm requests the Model for the necessary data, in the surface example, the color variable is accessible to them and MForm receives the message color sent by the VForm and CForm (interface components). The result of the user interaction is new surface appearance and a change of the internal state of MForm. CForm sends the message for the thickness update to the MForm, which provides methods for updating the model data. The CFSurface is the direct subclass of CForm and it is specialized for controlling the surface appearance. The CFSurface is the subclass of the Controller and handle the user input of the dialog box that accepts the new thickness parameter. Message passing mechanism requires the link from the controller to the model. Thus, the Controller cannot be predefined by the MVC framework, except the generic methods: Model and View. The controller concept is slightly extended with call-back messages, which are directly sent by a View to the Model ones the Controller has set the View reference. The design components of the MVC model are shown in Figure 186 for the Surface example and consist of the following three parts:

1. The surface is the application object, and it represents the Model.
2. The displayed surface on the screen is the View of the surface.
3. The input handling for the surface parameters is done through the Controller.

### 3.7 Visualization System Architecture

The MVC framework is applied for the visualization system architecture, which has its root Controller, View and Model components. The VS architecture can always be decomposed in these three components organized in a hierarchical structure, in which the explained surface example represents one leaf of that MVC composition framework with VS application at its root. The VS architecture is shown in Figure 187, where the main software layers are identified. The software layers re-group the classes in the following categories:

- **Input model category:**
  - Geometry group: base class Zone and its derivates for modeling of the structured and unstructured topologies
  - Continuum group: base class Field from which the Scalar and Vector fields are derived
  - CFD group: base class CFD object from which the input data classes are derived composed of Project, Mesh, Domain, Boundary and Segment classes.
- **3D Model category:** consists of classes that filter the Input data model and define different run-time models for 3D graphics, derived from the base class MForm for 3D scene building and 3D windows manipulation.
- **3D View category:** consists of classes that defined the appearance of the 3D graphics models and windows, derived from the base classes VForm. This category is enriched with AForm classes, which are responsible for controlling the appearance parameters of the displayed models.
- **3D Controller category:** consists of classes that defined the interaction control of the 3D graphics models and windows, derived from the base classes CForm and CWindow.
- **2D GUI category:** consist of base classes Button, Dialog and Menus which builds the 2D GUI layout. They are applied in as standalone components, as they are reused from existing libraries like Interviews and Tcl/Tk are.
- **2D & 3D Event handling:** is an important category of classes which synchronize Events, which the system generates from user interactions, and received from 2D GUI or 3D controller category.

The layers are described in the following Sections.
Figure 187: Visualization system architecture
3.7.1 Model Layer – CFD, Continuum and Geometry classes

The Input Model layer is constructed from CFD, Continuum and Geometry category of classes. The CFD classes organize the user input, which is modeled with geometry classes for structured, unstructured, and hybrid (their combination) topologies, and together with Continuum classes treats scalar and vector fields in 2D and 3D geometries. The CFD classes are modeling the problem domain including functionality for:

- Data management
- I/O operations
- Used in visualization algorithms

The input model consists of Project, Mesh, Domain, Boundary and Segment classes. Except the Project class they can be all specialized in different topologies (structured, unstructured) and defined in different dimensions (1D, 2D or 3D). The main idea regarding this decomposition is that each project can have multiple meshes, each mesh multiple domains, each domain multiple boundaries, and on each boundary multiple segments can identify different boundary conditions. For example, the solid boundaries are extracted at the initialization of the visualization session by this classification mechanism. The ObjectCFD class is the base class of this hierarchy and contains the indexing and labeling parameters to support such identification process. When the input file is read in, these set of classes take care to check the input correctness. They are the data container for all the visualization algorithms as they group the geometry and continuum classes to store the coordinates and quantity field data.

The geometry data are process with the classes derived from the base Zone class. They are Point, Curve, Surface and Body as modeled in chapter 1, which are then included in Quantity based classes such as Scalar Surface and Vector Curve. All these specialized application classes are including the foundation classes as Dictionary, Collection or Matrix classes for the storage of the numerical arrays and labeling lists.

For example, the Surface class defines the state and behavior for all subclasses like: SurfaceS or SurfaceU, defining respectively structured and unstructured geometry. The Surface class defines the interface that the graphics classes can access data for creating the surface Representations. With such design the manipulation of heterogeneous set of structured and unstructured surfaces will be invoked through the same interface message in order to refresh their graphical appearance. The designed common behavior at the abstract class level enforces the software components consistency in treating a variety of different implementations applying the same syntax. In addition, the memory storage requirements are reduced because the grid topologies store a cell-node relationship and all in between topologies (edge, faces) are calculated when requested.

As discussed, the ObjectCFD class is the generalization of the basic common data sets coming from CFD computation including geometry and related quantity data. The Field class associates geometry with a quantity through the Zone class. The ObjectCFD is usually composed of many quantities, primitive variables or derived quantities which are user selected for visualization. These values imply that the same geometry supports different number of quantity fields. For example, the density, momentum and the energy are calculated at the same grid nodes. The redundancy of geometrical information becomes obvious, when one is storing each field with duplicating its geometry. The ObjectCFD class is avoiding such a possibility by application of the referencing or sharable object mechanism.

For example, the Domain is always bounded by its boundary. The Boundary knowledge gives different opportunities for optimizing the search, when traversing the geometry or the quantity in order to extract different SubZones. SubZones can be extracted by indexing: 0, 1, 2, ...n; or quantity name can be applied for their identification. At insertion, all the processed ObjectCFD instances are checked in order to eliminate duplicates, as only one instance of the object can exist, to refer to key defined by an index and quantity name. The ObjectCFD class defines the automatic naming convention, which labels the ObjectCFD instances with a unique...
identification string. The most complex coding involves the specialized parts of the geometrical space decomposition modeled with the *Segment* class defining the boundary condition applied to that geometry. The input model is designed to support flexibility for processing of multiple components of each of the mentioned classes. For example, the labeling for of a segment could look as follows:

```
M1.D2.B3.S4
```

which means that we are identifying the segment 4 of the boundary 3 in the domain 2 of mesh 1. Each boundary consists of different BC segments, which can be further connected or not. In order to model such cascading relationship, the base *ObjectCFD* class, has a *Set* attribute to handle such parent-child relationship. These classes are also referencing the *Representations*, which are created during the interactive session, such as cutting planes and isosurface instances. For these instances the specific quantity field is only created, if the user selects that quantity for investigation; on which bases the creation of such field creation is triggered. The regeneration mechanism is implemented through the inheritance mechanism, which supports the invocation of a specific regeneration algorithm at the requested level, where the parameters for their creation are known. The recursive search is possible through the mentioned parent-child relationship.

![Figure 188: Hierarchy of Geometry classes](image)

### 3.7.2 View Layer – 3D Graphics Category

The *View Layer* is composed of classes which are designed to encapsulate 3D graphics, available through *application programming interfaces* (API) for accessing 2D and 3D graphics functionality. 3D Graphics category standardize a wide range of graphics features, which are available through different implementation of graphics standards such as *PHIGS, PEX, HOOPS* and *OpenGL* implementations, see Figure 189. This graphics layer assures 3D graphics portability to a wide variety of computer platforms ranging from personal computers (PC) to workstations, to supercomputers. The *View Layer* classes are responsible to generate 3D graphics, which is constructed from data received from the *Model Layer* and sometimes, additional parameters from the attribute
classes (part of the View Layer), which are manipulated through the Controller layer in order to provide appearance parameters of the selected graphics mode. The View Layer provides a visualization system with more adapted functionality for building 3D scenes, navigation and windowing operations. It supports 3D graphics content composition, display, and interaction. The graphical primitives for building the Representations are characters, lines, polygons or other graphics shapes available from the underlying 3D graphics engine. The 3D graphics classes for the modeling of an interactive visualization system are expected to support the following capabilities:

- geometric and raster primitives,
- RGBA or color index mode,
- display list or immediate mode (tradeoff between editing and performance),
- 3D rendering:
  - lighting,
  - shading,
  - hidden surface, hidden line removal (HLHSR), (depth buffer, z-buffer),
  - transparency (alpha-blending),
- special effects:
  - anti-aliasing,
  - texture mapping,
  - atmospheric effects (fog, smoke, haze),
- feedback and selection,
- stencil planes,
- accumulation buffer,
- compatibility, inter-operability and conformance-compile with different OS platforms like: DEC, IBM, SGI, HP, SUN, PC-Windows, PC-Mac etc.

Figure 189: 3D View Layer
The graphics API accesses the graphics hardware to render 2D and 3D objects directly into a frame buffer. These objects are defined as sequences of vertices (geometric objects) or pixels (images).

**Graphics primitives** are:
- points,
- polylines,
- polygons,
- bitmaps.

They are defined by a group of one or more vertices (nodes). A vertex defines a point, an end point of the polyline or a corner of a polygon, where two edges meet. The vertex may consist, apart from the geometrical coordinates, of: color, normal, texture coordinates and edge flags.

**Fundamental operations** involve:
- transformation matrices,
- lighting equation coefficients,
- anti-aliasing methods,
- pixel update operators.

**Rasterization** is the process by which a primitive is connected to a 2D image. Each point of this image contains the information about color, depth and texture. **Frame buffer** consists of logical buffers for:
- color,
- depth,
- stencil,
- accumulation.

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Rasterization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinate transformation</td>
<td>Pixel operation</td>
</tr>
<tr>
<td>Coloring and lighting</td>
<td>Texture Mapping</td>
</tr>
<tr>
<td>Clipping</td>
<td>Evaluators</td>
</tr>
<tr>
<td>Primitives</td>
<td>Rasterization</td>
</tr>
<tr>
<td>Geometry</td>
<td>Graphics Primitives</td>
</tr>
<tr>
<td>text</td>
<td>string</td>
</tr>
<tr>
<td>points</td>
<td>marker</td>
</tr>
<tr>
<td>curves</td>
<td>polyline</td>
</tr>
<tr>
<td>surfaces</td>
<td>polygons, fill area facet</td>
</tr>
</tbody>
</table>

**Transparency** is best implemented by using blend function. Incoming (source) alpha is correctly thought as a material opacity ranging from 1.0 (representing the complete opacity) to the complete transparency. In RGB mode is appropriate, while in Color index mode is ignored.

**Graphics primitives** can be used to create forms as shown in Table 32. Graphics attributes of such Forms consists of variety of colors and rendering styles. They are applied to localize and analyze rapidly geometric entities.

<table>
<thead>
<tr>
<th>Geometry</th>
<th>Graphics Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>string</td>
</tr>
<tr>
<td>points</td>
<td>marker</td>
</tr>
<tr>
<td>curves</td>
<td>polyline</td>
</tr>
<tr>
<td>surfaces</td>
<td>polygons, fill area facet</td>
</tr>
</tbody>
</table>

**Graphics attributes** are:
- color - color, - color map,
- paint option
- culling,
- refinement.
Window attributes are:
- color,
- ambiance light vector
- z clip depth,
- eye begin, end.
- up vector
- window view
- view box
- view plane

The 3D Graphics Category of classes is the layer that supports the display of representation objects created by the visualization system. It provides a support for
- window creation,
- manipulation of window properties,
- editing of graphical attributes associated to geometric entities.

A separate window has to be attached to the graphics object to select it for manipulation and the update has to be sent to the graphic group. Such behavior requires a tighter integration between the graphic group and the graphic part to handle presentation and interactivity. Graphical groups and their parts coordinate their actions in order to achieve the correct display. These actions are split into three parts:
- rendering,
- layout negotiation (scaling, sorting),
- screen update mechanism.

The important operation is the request for a Form updates on the screen, when being part of a group. The group defines the order in which to traverse the group parts independently from the operation being performed. One characteristic operation is the finding of the Minimum fitting box. Form objects do not update the screen directly, but they request the redraw operation on them, which put them in a ready state in order that the update can be performed, as the updating of a complete screen involves redrawing of all displayed objects (views). If the Form is not present in the exposed region, it will be not drawn. Deferred updates’ effects are:
1. the clipped region can affect a Form several times and in a direct update mode, it would be needed to be refresh twice.
2. double-buffering optimization,
3. clipping of complex objects.

3.7.3 Controller Layer – 3D and GUI input processing

The Controller layer is responsible for processing the user interaction with the visualization system. The Controller classes are modeling the control of the input devices like mouse and keyboard. The 3D and GUI objects, which are displayed, might be selectable through the user interaction, and in that case they have an associated controller for handling the expected user request. The important controller classes are CWindow, CView and CSurface, as shown in Figure 190.

Controllers are grabbing the user input and through the input handling mechanism, they receive the necessary information to trigger Actions, which usually access respective models and update their views on the screen. An example for CWindow control is the management of views, while CView controls the displayed surfaces. CSurface controls different rendering modes of a specific selected surface. These controllers are related to the 3D graphics engine, while in parallel we also need to control the input coming from the 2D GUI components, like menus and buttons. They also provide the interface between input devices (for example: keyboard, mouse and timers) and the associated model and view layers. They do not only deal with different input devices but they also provide higher level activity, such as mouse tracking, scheduling of user’s requests, window resizing and movement event. The controller layer objects define the protocols of the user’s interaction with the model layer, and they provide methods between the externally generated events and the display elements modeled by the view layer. The developed GUI components, as presented in chapter 2, where designed through several
iterations before the final design was reached. They control the data input, and provide feedback on the user interaction, including errors reporting. The feedback information is extremely important as it keeps the user informed about the system activities. The **GUI** toolkit contains general components for designing the user interface, which cover the following functionality:

- presenting menus,
- parsing commands
- reading free-format numeric input,
- handling text input,
- presenting alerts and dialog boxes,
- displaying windows,
- presenting help, and
- outputting user provoked errors

The **GUI** combines all the three kinds of menu components: *menubar*, *pulldown* and *pulldown* menus. The menu items can have shortcut commands associated to the keyboard input for speed-up the interactions of experienced users. The visualization system is a command-driven system and operates by sequentially processing the triggered commands and their associated parameters, executed through the *Action* hierarchy of classes. The **GUI** classes have been designed to support a common interface to 2D **GUI** elements, which implicitly take care of the window system platform by wrapping the needed functionality to assure:

- Intersection of functionality → it is common to all the windowing systems.
- Union of functionality → all the windowing system functionality is encompassed.

**Window** as a base class supports different kinds of user’s windows, such as:

- *application window*,
- *icon*,
- *warning* and
- *dialog.*

![Class hierarchy of the controller classes](image)

*Figure 190: Class hierarchy of the controller classes*
System is a main controller manager that synchronizes the input-output control of the application window consisting of three parts:
1. menu bar and time header,
2. working area for placing controllers,
3. monitor area with toolbox controlling the current active view and automatically assigning view manipulation tools.

The System class coordinates these two different events handling mechanisms in the unified form by taking the case of synchronization and event processing order. This activity is tightly linked with the update mechanism and especially important for 3D graphics updates. The System delegates to the Display class the responsibility to activate updates on the specified views. The user operation is not associated directly to a particular user interface, especially important for 3D graphics updates. The Menu Item is responsible to handle the user generated event, which contains the user request. The Event is captured by the event Dispatcher, which sequentially process them. The Menu Item is associated with Action, which is executed when the user selects a Menu Item and it is responsible to carry out the user request. The user request involves an operation associated with different combinations of objects and operations. The parameterization in the Menu Item is done through the object Action. The Action abstract class provides an interface for issuing a request. The basic interface consists of a unified execute message, which is propagated through the inheritance mechanism, till the requested specialization is found. The Pulldown menu is an example of the class that triggers an Action in response to the button down event. The buttons and other visual interaction components are derived from the Controller - Input Handler classes, which associate Actions in the same way as done for the Menu Item class.

![Event/Action coupling](image)

The undo ability is the characteristic of the action that can cancel its own effect. The addition of the unexecute function to the Action interface allows such features. Sometimes the undo ability is determined at run-time. The final step in supporting the arbitrary-level of undo/redo is defined in the Action history, which is traced through the Macro class. The number of undo levels is limited to the command history. The command pattern prescribes a uniform interface for issuing the undo request. A command may delegate all, part or none of the requests to other objects. The orientation towards the visible objects’ results in more error tolerance because every controller class contains operations that make reuse in its own framework. The connection of the interaction object with the model is supported with call-back messages. The Controller class is slightly extended with call-back concept applied for the controller messages. The messages are directly sent by controller to the model, as a reaction to the action. The model receives the message, and triggers its views to update.
3.8 Software Development Environment

3.8.1 Software Reusability

The developed software application and class library provide reusable components to support the software development of a scientific visualization system based on the C++ programming language. The software design itself can be reused in building new visualization application, in another object-oriented programming language, like Java, as it shows clearly the visualization components and their interactions. The generic layer of the designed application consists of abstract classes, defining functionality for each major component and its related interface protocol in terms of messages. Abstract base classes are representing common data and procedures associated to particular entity. The abstract class message protocol defines also relationships with other classes. Message’s protocols of the implemented classes might be interchanged with the ones declaring the same message interface. This is particularly interesting for the GUI components, when adapting the visualization system for other application domains.

3.8.2 Software Portability

The 3D rendering techniques are platform dependent and the performance issue needs to be carefully tested. The graphics library offers computer and device independence, but it needs to be verified on different computer systems the involved output devices:

- graphics hardware
- terminals
- plotters
- laser pointers

Portability is an important condition in order that the visualization system is running on different UNIX based workstations and PC (Windows, Mac). Therefore, when CFView was being designed to be run on a variety of different graphics systems we need to consider the following elements:

- Number of available colors.
- Line width support and availability of line type extension.
  - Polygon fill - maximum number of vertices availability, of pattern and hatch fill ability, to fill certain classes of polygons.
  - Text (hardware character sizes and characteristics)
- Selective erase
- Size of display (input echo area).
- Interactive input support
- Double buffering,
  - Z - buffering,
  - Lighting & shading.

3.8.3 Integrated Development Environment

An object-oriented Integrated Development Environment (IDE) needs to be put in place in order to improve the visibility of the source code and control of its evolvement, for example Eclipse or Microsoft Visual C++ in order to handle the application building (compiling, debugging and testing). An application, as the one presented in this thesis, consists of several hundred modules, which need to be systematically compiled with different options. Such IDE keeps track of all the necessary files and their interdependencies, as the software development project consist of many files, which are depending on each other, and organized, in different directories.
structures. Some files are source code (headers *.h, and source *.c) while others are object files *.o, produced by
the compiler and executable ones by the linker. The last one depends directly on the source code.

An O-O development environment supports the sense of design. An application is decomposed in terms of
classes and their interactions. They are coded in C++ code and linked among several directories, which contains
a number of files, which needs careful and disciplined organization and naming convention. The source code is
divided in two major parts. The include-files are named <class name>.H and define the most abstract layer of
the application. They describe the class interface protocol: class declaration defined in terms of messages. The
source-files, which define the definition of the messages, are named as <class name>.C.

In addition, the development environment must have a class-hierarchy browser which ties together all the C++
classes and maybe some native C or FORTRAN code. The visual presentation of the class hierarchy is an
important aid to the developer, to make him/her easy to comprehend, the applied (sometimes complex)
inheritance and polymorphic model. A class browser allows to access and to edit, in highly structured way the
class files and the executables. It needs to provide menu-driven operations to import existing classes into a
project create new classes and delete existing classes from an application. The facility to visually create new
number functions, data members and friends, to examine, modify, delete existing members, to change the
visibility of any member and to change a member function to be / static, inline / normal or friend represents the
indispensable tool for modern software development. The makefile manages the mechanics of building the
application by keeping track of many included files required in C++ program. The automatic construction and
maintenance of makefile which controls the dependencies of the compile and link process is essential to be
integrated in IDE. The benefits from the use of such development environment are:

- significantly reduced design and development time due to high degree of sensibility, extensibility
and ready to use components,
- significantly reduced program size,
- enforced consistency of the GUI which have feel-and-look characteristics built in,
- enforced portability because all system and device dependencies are eliminated through the property
of encapsulation and unified protocols.

A computer program is a complex artifact usually presented as text. Visual presentation of code is a method of
presenting program structure visually. The program visualization is concerned with the following aspects:

- techniques used to display a program visually,
- how visualization affects the software development and maintenance,
- general features of program visualization tools.

The graph representation is standard visual technique to present the program structure. The graph nodes
represent program entities such as classes and functions, while graph curves represent the relationship between
these entities. For example: Inheritance and function calls. Using different line styles or colors several
representations can overlay on the same display or window or image. The C++ language provides a rich set of
syntax, which can be visualized as graph nodes. That includes classes, class templates, class members, functions,
function templates, variables (both local and global) and files together with the relationship between them.

- Inheritance which different types of visibility:
  - public
  - private
  - protected

- Containment or Aggregation. - One class contains another class as:
  - data number,
  - instance,
  - pointer,
  - reference.
- Instantiation via template
- Friend members
- Function call: relationship between class variable nodes:
  - direct function call (is a relationship when a function directly calls another function),
  - indirect function call (it exists when a function calls another via a function pointer stored in the pointer),
  - virtual function call (exists when a function calls a virtual function of defined class,
  - global & local variable,
- Uses and modifies. This relationship exists between variable nodes as structure nodes and function nodes. The arc indicates that the function either references or changes the value of the variable. When the relationship exists between a function and a structure, it indicates that the field of the structure is either referenced or modified.
- Definition of variable, function and class
- Declaration of include dependencies

The program visualization techniques provide a high level view of program structure. The benefits of such approach help reusing of the elements and improving by adding libraries, to be integrated as basic structure of the application. The IDE tool has to consist of:
- source code versioning
- class hierarchy and class relationship diagrams for browsing and picking the source code
- data flow diagrams
- editor with syntax identification through different font types and colors
- output of subparts of the code with associated diagrams in appropriated graphics format
- performance benchmarks and quality tests.

![Figure 192: Eclipse Integrated Development Environment](image-url)
**Conclusion and Future Developments**

Innovation in visualization systems poses the simultaneous challenges of: building better, faster and cheaper computer-aided solutions to ever more complex scientific, engineering and other multi-disciplinary problems; developing sophisticated methodologies and algorithms; harnessing the power of upcoming technologies; reusing and leveraging the power of legacy systems and solutions; and working in increasingly shorter design and production cycles.

The work presented in this thesis has been addressing, over many years, the problem of advancing the state-of-the-art of scientific visualization systems using innovative software development methods and programming techniques, specifically object-oriented methodologies and tools. We have succeeded in demonstrating that object-oriented approaches and techniques are appropriate for designing and building interactive visualization systems that meet all the requirements placed on them by scientific disciplines: correctness, accuracy, flexibility, performance and by software engineering: compatibility, reusability, portability. In particular, we have shown how a high degree of interactivity and user-friendliness can be achieved through the use of class libraries encapsulating graphics and interfacing. More importantly, we have provided evidence that scientific visualization has deeply changed the very nature of the investigative process itself by allowing the researcher to explore and view the physical world in an intuitive, interactive and deeply illuminating manner.

We have illustrated our approach with examples taken from the genesis and development of CFView, an advanced interactive visualization tool capable of handling complex scientific and engineering problems, in various geometries and in virtually any application/investigation area. CFView -- and its several prototyping variants produced over the years -- is the result of many iterative developments in continuous interaction with engineers and scientists, mostly but not exclusively at the VUB [71, 111-124]. Many PhD theses have been produced in the context of the research, design, development, prototyping, testing and packaging work that took place in relation to the CFView system.

The process of transforming user requests into software functionality has been analyzed and mastered at all detail levels, leading to a Graphical User Interface very-well adapted to the requirements of CFD engineering techniques, as demonstrated by the commercial success of the CFView product marketed by NUMECA. The software architecture, a main result of the present work, has proven to be flexible enough to cope with the increasingly demanding visualization needs of advanced industrial turbo-machinery applications in ongoing development at NUMECA.

![Knowledge domains involved in interactive visualization](image)

*Figure 193: Knowledge domains involved in interactive visualization*
In order to provide for the interactive use of many different functionalities, a visualization system needs to integrate several types of components. The graphical user interface, for example, which provides user interactivity, relies on a dynamic windowing environment together with several I/O facilities and sets of graphical primitives; the visualization operators are another example of software built upon code originating from computational geometry \cite{125} and combinatorial topology \cite{126}. The number and the diversity of the components/routines that must be handled and their tight couplings add considerable complexity to the task of developing and maintaining visualization software over an extended period of time. To deal with this complexity, our software was developed using object-oriented programming, allowing different data types and routines to be encapsulated in ‘objects’. These objects are grouped in classes covering different application domains. The classes themselves are organized into class hierarchies, which match both the internal knowledge domain and link domains in-between, see Figure 193. Using OOP, one can systematically and naturally decompose complex software into manageable modules.

As shown in Figure 193, interactive visualization calls upon various knowledge areas and conceptual models, namely: application data model, computational geometry, combinatorial topology and computer graphics. The application model is meaningful to the scientists who want to be able to identify, extract and view regions of interest in the application model. Computational geometry and combinatorial topology provide the set of operators needed to carry out the visualization process; these operations are specialized mathematical transformations and mappings that operate on the application data sets. Finally, computer graphics provide for the concepts that relate to the user interface and for the graphical primitives that display data on the monitor screens.

*Scientific Visualization* has become a feature intrinsic in the scientific investigation and discovery process, indeed a methodological instrument of scientific research. *SV* instruments help users to gain novel, often revealing insights into complex phenomena. The investigator controls *SV* tools to ‘jump’ from phenomenon overview to detail analysis then back from details to overview, with as many iterations as required to gain a better understanding of the phenomenon under study. *Interactive* visualization, with its ability to integrate and synthesize many rendering/viewing techniques and manipulations, improves the perceptual capabilities of the user in a simple, intuitive and self-explanatory manner.

The main features of the software we have developed include:

- Integrated structured and unstructured geometry treatment for efficient visualization of numerically-generated data around complex 2D/3D geometries,
- Visualization environment with interactive user control,
- Development of a highly portable graphical user interface (for easy access to integrated system components),
- Creation of class libraries, as reusable and extensible components for different software products,

We have investigated visualization methods covering:

- Transparent 2D/3D interactive visualization tools with user-controlled parameters.
- Interactive 2D/3D qualitative tools, local values, sections, arbitrary cutting planes.
- Interactive techniques simulating smoke injection (particle paths).

The interactive techniques comprise:

- Interactive creation of non-computational grid surfaces (cut planes, iso-surfaces) where analysis tools can be applied (e.g. sections, isolines),
- Interactive identification of block topology and connectivity with boundary conditions,
- Analysis of multiple quantities related to the same geometry in the same or side-by-side view,
- Data comparison in a multi-view, multi-project environment.

The visualization tools were integrated in the GUI in order to enable seamless communication and control between different geometries and quantity representations.
In the following section, we describe the impact that our visualization system has had on 5 EC and IWT R&D projects.

**PASHA – Parallel CFView – improving the performance of visualization algorithms**

Parallel CFView, developed under the ESPRIT III-7074 PASHA project, is a heterogeneous, distributed SIMD/MIMD software system whose concept is illustrated in Figure 194. The system is divided in 3 independent components:

![Figure 194: Conceptual overview of the SIMD/MIMD Parallel CFView system](image)

1. The CFView visualization package, which runs on a high-end graphical workstation.
2. Parallel Applications that run on parallel machines. These applications can be viewed as a collection of stand-alone parallel programs performing mapping operations on CFD data.
3. The Interface Framework, which governs the communication between CFView and the Parallel Applications. The Interface Framework is a distributed program which for its largest part runs on a Parallel Server machine, but with extensions to the graphical workstation and the parallel computers. Among its tasks are communication management, parallel server control, networking and synchronization. It provides generic functionalities for transparent access to the parallel applications in a heterogeneous and distributed environment.

The four algorithms in an SIMD (as well as in an MIMD) implementation of Parallel CFView system [127] are:

1. The `parallel cutting plane` algorithm uses a geometrical mesh, a scalar quantity and the equation of a plane to calculate a collection of triangles and scalar quantities. The triangles represent the triangulated intersections of the plane with the mesh. The scalar data are the interpolated values of the scalar quantity at the vertices of the triangles.

2. The `parallel isosurface` algorithm uses a geometrical mesh, a scalar quantity and an isovalue. It calculates a collection of triangles. The triangles represent the triangulated intersections of the isosurface with the given isovalue with the mesh.
3. The **parallel particle-tracing** algorithm uses a geometrical mesh, a vector quantity and the initial positions of a number of particles in the mesh. It calculates the particle paths of the given particles. The paths are represented as a sequence of particle positions and associated time steps.

4. The **parallel vorticity-tracing** algorithm uses a geometrical mesh, a vector quantity and the coordinates of a number of points in the mesh. It calculates the vorticity vector lines associated with the vector quantity that pass through the given points. The lines are represented as a sequence of positions.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#triangles</th>
<th>System</th>
<th>Send Data</th>
<th>Load Data</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
<td>SIMD</td>
<td>3.66</td>
<td>5.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MIMD</td>
<td>3.66</td>
<td>28.66</td>
</tr>
<tr>
<td>Cutting Plane</td>
<td>5189</td>
<td>Sequential</td>
<td>SIMD</td>
<td>3.66</td>
<td>5.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MIMD</td>
<td>3.66</td>
<td>28.66</td>
</tr>
<tr>
<td>Isosurface</td>
<td>10150</td>
<td>Sequential</td>
<td>SIMD</td>
<td>3.66</td>
<td>5.82</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MIMD</td>
<td>3.66</td>
<td>28.66</td>
</tr>
</tbody>
</table>

*Table 33: Average times (s) for Sequential, SIMD and MIMD implementations of Cutting Plane and Isosurface algorithms (wall-clock time)*

In order to evaluate the performance of the heterogeneous and distributed approach for Parallel CFView, a limited benchmarking analysis was conducted. We compared the performances of the SIMD and the MIMD implementations with the stand-alone, sequential CFView system. The SIMD parallel machine used was a CPP DAP 510C-16 with 1024 bit processors and 16MB RAM (shared). The MIMD parallel computer was a Parsytec GCeI-1/32 with 32 T-805 processors each having 4 MB of RAM. Both parallel machines were connected (through SCSI and S-bus respectively) to the Parallel Server machine which is a SUN SparcStation10. For the graphical workstation, we used an HP9000/735 which communicates with the Parallel Server over an Ethernet LAN. The results of the measurements are given in Table 33. All times are in seconds, averaged over 20 runs. The table shows the average execution times for both algorithms on the different systems.

For the SIMD and MIMD Parallel CFView implementations, the average time needed to send the data (mesh and scalar quantity) from the graphical workstation to the Parallel Server is given (Send Data), as well as the average time needed for loading the data from the Parallel Server onto the parallel machines (Load Data).

The average execution time (Execute) for the parallel implementations includes (i) the sending from the graphical workstation to the parallel machines of the algorithmic parameters (i.e. the equation of the cutting plane or the iso-value); (ii) the execution of the algorithm on the parallel machines; (iii) the retrieval of the result data from the parallel machines to the Parallel Server; and (iv) the sending of the result data from the Parallel Server to the graphical workstation. For the stand-alone, sequential implementation of CFView, only the average execution time is shown. The number of triangles (averaged over the runs) generated by the algorithms is given.

The total execution time, given in Table 33, for the parallel implementations is the sum of the three timings (Send, Load, and Execute). However, by making use of the caching mechanism in the Interface Framework, data send and load needs be done once only, in the beginning. After that, only the new equation of the cutting plane or the new iso-value for the isosurface has to be transmitted to the parallel machines, after which execution can take off. Hence, in a realistic situation, only the times listed in the last column (Execute) are relevant for comparison.

The results revealed how massively-parallel computers (SIMD as well as MIMD) can be used as powerful computational back-ends in a heterogeneous and distributed environment. A performance analysis of Parallel CFView showed that both types of parallel machines are about equally fast. The total execution times on the SIMD implementations are sensitive to the amount of computation required, whereas the execution times on the MIMD implementations are dependent upon the amount of data routed between the processors. The overheads induced by the interface framework are seen to require only a minor fraction of the global execution times. This
indicates that the heterogeneous and distributed approach of Parallel CFView is indeed viable since it performs significantly better than its stand-alone, sequential counterpart. This is especially true for computationally-expensive operations, such as isosurface calculations on problems with large data volumes. Since the heterogeneous and distributed nature of the system allows the transparent use of remote parallel machines on various hardware platforms.

**Alice – QFView – towards the transparent visualization of numerical and experimental data sets**

The development of QFView in the ESPRIT-IV “ALICE” project (EP-28168) extended the author’s research towards using the World Wide Web for designing and building up distributed, collaborative scientific environments [47, 128]. QFView was developed in a web-oriented client-server architecture (e.g. Java, JDBC) which allowed openness and modularity, as well as improved flexibility and integration of the visualization components (current and future). A core element was the creation of a central database where very large data sets were imported, classified and stored for re-use. The distributed nature of QFView allows the user to extract, visualize and compare data from the central database using World Wide Web access. QFView integrates EFD and CFD data processing (e.g. flow field mappings with flow field visualization).

QFView supports the integrated use of visualization and animation tools which are integrated with the database management system for data access and archiving. For example, in a possible configuration of QFView components over the Internet, the J2EE application servers [129] and the Database Management System servers (DBMS) [130] are located in one site, whilst the GUI client applications run on a server in another location (see Figure 195 (a)). The meta-database is based on a J2EE distributed architecture whose execution logic is stored and executed at the EJB container level [131], and which is accessed from the Java based graphical user interface (GUI) via the HTTP protocol. The main advantage of the Internet is the possibility to store and access data (images, input files, documents, etc.) from any URL in the world. Both EFD and CFD applications generate huge quantities of data which can be used to populate the database. The user needs to access, extract, visualize and compare the required quantities; these functions as illustrated in Figure 196. The QFView system is composed of 3 major elements:

1. An “EJB container” with all the metadata management rules to manipulate metadata and the relational database used to store the metadata information. The EJB container acts as security proxy for the data in the relational database.

![Figure 195: QFView – an Internet based archiving and visualization environment](image-url)
A “Thin GUI Java client” is used for remote data entry, data organization and plug-in visualization. GUI clients must be installed at the end-user location, either at application installation time or by automatic download (Zero Administration Client).

URL accessed data (images, video, data files, etc.) can be placed at any URL site.

QFView organizes, stores, retrieves and classifies the data generated by experiments and simulations with an easy-to-use GUI. The data management component, see Figure 195 b, offers a user-friendly web-enabled front-end to populate and maintain the metadata repository. The user can accomplish the following tasks using this thin GUI Java client:

- Start the GUI Java client application, create a new folder (measurement), define metadata information for the folder, such as keywords, physical characteristics, etc. It is important to emphasize that the GUI Java client is connected to the EJB server using the HTTP protocol, and all the information entered by the user are automatically stored in the relational database.

- Organize the data into a hierarchy of predefined or newly-created tree-like nodes; user can also execute a data search procedure, combine documents, and do several other operations on the input folder.

- Create and define new raw data such as XML files, images, input files, etc. for a particular folder by specifying either a local or a remote location of the data.

- Define old raw data (XML files, text files, videos, etc.) by specifying either a local or a remote location of the data.

- Start the visualization plug-in application on the selected raw data.

Essential Database Tools present in QFView are:

a) Data Entry
b) Folder and Document insertion
c) Data Classification
d) Full Text Search
e) Meta Model and Data Organization
These tools were used to assist experimental setup, to improve data acquisition and systematic manipulation of the results of the VUB’s double-annular jet modeling (Figure 197, frame 1 from left) of the flow in a prototype combustion chamber. The requirement was to analyze the flow from LSV data (frame 2), PIV data (frame 3), LDV data (frame 4) and CFD calculation data (frame 5). The LDV (frame 4) clearly shows the mean flow field, which can be compared with CFD (frame 5). Qualitatively they produce a similar topology but the prediction is not accurate. For correct prediction of the flow, it is necessary to take into account the effect of coherent structures (frame 3), which strongly influence the combustion process [132].

The QFView system coupled to the unified numerical/experimental database enables:

- the users to reduce the time and effort they put into setting up their experiments and validating the results of their simulations,
- the technology providers to develop a new products [133] capable of meeting evolving and increasingly demanding industry requirements.

The users have observed that QFView provided them with a means not only for archiving and manipulating datasets, but also for organizing their entire work-flow. The impact of visualization systems like QFView on the investigative process itself opens the way to entirely novel ways of working for researchers in experimental and computational fluid dynamics. Integrated, distributed, collaborative visualization environments give the possibility of and indeed point to the need of reorganizing research methods and workflows; the various ‘experiments’ conducted with QFView in ALICE have given glimpses of the exciting advances that one may expect from such systems in the coming years.

**QNET-CFD – on quality and thrust in industrial applications**

QNET-CFD was a Thematic Network on Quality and Trust for the industrial applications of CFD [134]. It provided European industries with a knowledge base of high quality application challenges (reviewed and approved CFD results) and best practice guidelines. QNET-CFD was part of the EC R&D GROWTH program. Eight newsletters see Figure 198 and four workshops on Quality & Trust in CFD were coordinated by VUB during the project.

The main objectives of the project were:

- To create the Knowledge Base to collect Application Challenges and Underlying Flow Regimes from the trusted sources and made them available to the Network members;
- To publish 8 issues of the Network Newsletter to report on the Network’s activities
- To maintain an open Web-site to publish the Network’s work progress and achievements
- To organize 4 Annual Workshops, a key instrument for disseminating material on advances and achievements in Quality & Trust, validation techniques and uncertainty analysis.
Figure 198: The eight QNET-CFD newsletters

The project showed that making trusted and validated computational and experimental data sets widely available is of great importance to the EU scientific community; also, that the integrated processing of CFD and EFD data requires appropriate access and manipulation tools. As visible on the cover pages of the QNET-CFD newsletters, scientific visualization has become the most effective and striking way of analyzing presenting scientific data; obviously, new SV tools will be required for interactive visual access, manipulation, selection and processing in the future, and such tools can be built upon the basic QFView architecture pioneered in the ALICE project.

**LASCOT –Visualization as decision-making aid**

The LASCOT project [55] is part of the EUREKA/ITEA initiative. The Information Technology European Advancement (ITEA) program for research and development in middleware is jointly promoted by the Public Authorities in all EU Members States and some large European industrial companies.

The goal of LASCOT was to design, develop and demonstrate the potential benefits of distributed collaborative decision-support technology to the “future cyber-enterprise in the global economy”; the LASCOT demonstrator was to:

- Support access to traditional information systems and to Web data;
- Enable situation assessment, and provide decision-support facilities as well as simulation and validation facilities to support business decisions;
- Include current, enhanced-as-required security tools;
- Make use of visualization technology for critical tasks such as decision-making and knowledge management;
- Produce an on line learning application to facilitate the embedding of the platform, by the users.
The scenario that was retained to demonstrate the LASCOT system is illustrated in Figure 200; it shows the various ‘actors’ who are aided by the LASCOT system in monitoring a crisis situation and in making decisions. The project called for 3D visualization research and for the development of 3D presentation tools capable of treating general-purpose and highly conceptual information in an appropriate and readily understandable manner. In this project, our research was focused on the visualization and manipulation of graphical content in a distributed network environment. Graphical middleware and 3D desktop prototypes [135] were specialized for situational awareness, see Figure 199. A state-of-the-art review did not identify any publicly available large-scale distributed application of this kind. The existing proprietary solutions rely on conventional technologies and are limited to 2D representation. Our challenge was to apply the latest technologies, such as Java3D, X3D and SOAP, compatible with average computer graphics hardware and to demonstrate a solution allowing: data flow from heterogeneous sources; interoperability across different operating systems and 3D visual representations to enhance the end-users interaction.

We applied the Model-View-Controller (MVC) paradigm to enhance the interactivity of our 3D software components for: visualization, monitoring and exchange of dynamic information, including spatial and time-dependent data, see Figure 199. The software development included the integration and customization of different visualization components based on 3D Computer Graphics (Java3D) and Web (X3D, SOAP) technologies and applying the object-oriented approach based on Xj3D.

Cutting-edge 3D graphics technologies where integrated, including the Java-based X3D browser; we used Xj3D to visualize data from various external sources using our graphical middleware. While present software components provide highly flexible interactions and data-flows, the coupling between these components is designed to be very loose. Thus, the components can be upgraded (or even replaced) independently from each other, without loss of functionality. With SOAP messaging, inter-components communication is completely independent of software platforms and communication transmission layers. In our approach, Java components co-exist with Microsoft .NET front-end, as well as back-end implementations. This approach allows improving software development of 3D collaborative and visualization tools. Future development of an appropriate ontology could significantly improve the distributed visualization framework.

Figure 199: The LASCOT application
The Actors/Stakeholders that have partial information inside their information systems provide input to the LASCOT System via SOAP/XML Middleware.

LASCOT Dynamic Business Process Engine runs the business process workflows based on Decision Scenarios.

Decision Scenarios create/drive the business process workflows.

Graphical Middleware that transforms the Common LASCOT XML information objects into data sets suitable for the visualization engine.

Visualization Engine presents information as 2/3D visual objects and alerts “problem areas” to the crisis manager and other stakeholders.

Figure 200: The LASCOT scenario
The SERKET project [136] intends to explore a solution to the issue of security in public areas and events by developing an innovative system whereby dispersed data from a variety of different devices are automatically correlated, analyzed and presented to security personnel as ‘the right information at the right time’. The aim is to design and to develop an open-software platform that can be deployed at low cost.

3D software development in SERKET is centered on the visualization and presentation engine, with special attention on the application of X3D (and XML) standards. The graphical middleware must integrate, correlate, combine, annotate and visualize sensor data and related metadata (the application context is airport security). Using sensor data analyzed by other processing and data fusion components, the graphical middleware will build 3D scenes which represent the objects detected by the sensors and the operational status of the sensors at their locations. Objects in the 3D scenes will be annotated with metadata and/or with links to metadata describing the security context in relation to the displayed 3D objects. The 3D rendition of the situation must provide an unambiguous, highly understandable overview of the situation to the user, who should be able to switch between different levels of viewing details and select, at each level, desired viewpoints (the locations of the video cameras define the available viewpoints). The 3D model of situation-security awareness will be parameterized in space and time as shown in Figure 201.

The solution will be based on the J2EE Application Server platform (JBoss). The interface between the components will be based on SOAP messaging. 3D objects and scenes will be using the X3D file format (an extended XML-based format for storing graphics). The XML mark-up language will be used for data exchange. X3D can be extended to include meta-data, thus non-graphical information related to the created graphical object. The development for the graphical middleware visualization components will be done in Java, using the Eclipse environment. The 3D SERKET desktop will enable access to and treatment of security data from heterogeneous sources and will support:

- 3D models with features
- Level of Detail Selection
- Navigation and Interaction

Figure 202 illustrates the main functionality of the SERKET application; clearly, 3D interaction and 3D models are key elements to a new generation of visualization software which must now treat and be able to display general-purpose and abstract information such as ‘security’.
Future developments

Web3D for Collaborative Analysis and Visualization

Today’s computer graphics technology enables users to view spatial data in a ‘true-to-life’ spatial representation, to the point that non-technical people can comprehend complex data with easiness and comfort. Visualization tools and data are now the most common media for people -- both technical and non-technical -- to exchange information. Yet, not everyone has access to visualization tools, although access to the Internet keeps increasing as the public becomes more at ease with the idea of using the Internet-based services. To meet all demands for visualization and make the best use of existing and upcoming ICT, we need methodologies and tools that can be applied collaboratively to collect, inventory, process and visualize 3D data provided by possibly thousands of servers in the world. A main requirement is that such Web3D-enabled visualization will occur interactively, on demand and in real-time (“seeing is believing”).

The challenges are enormous and research will be about creating new visualization tools and combining them with existing ones to 3D computer graphics and Web-based technologies, and about developing collaborative, interactive 3D analysis and visualization methodologies for building real-time virtual reality (VR) worlds. We need visualization aids capable of combining and interpreting Web-based spatial and temporal data to assist us in the analysis of shared information (supporting collaboration) over the Internet. The two main user activities on the Web -- content retrieval and content viewing -- will be profoundly modified in this Web3D approach by the Client Spatial Interpreter (CSI), a new component which will enable the users to perform spatial analyses of contents retrieved and viewed with the help of Graphical Middleware (GM). A first step will be to reconstruct spatial data with a conventional PC and interactive 3D visualization using object-oriented computer graphics. The next step will be to extend the previously mentioned application for collaborative analysis and visualization,
with the use of VR technologies, of Web3D, VRML/X3D and Java3D, Internet Map Server (IMS), Server Pages (ASP and JSP) and Spatial Database Engine (SDE) and Graphical Middleware (GM). These technologies need to be coupled with application knowledge and to use the 3D web for viewing it, as 3D spatial analysis can be carried out by an Internet user in real times with the data coming from different servers.

**Problem Statement:** “Although Spatial Analysis and Modeling can be achieved on the desktop using Visualization Software tools, the development of a Methodology for Web3D Collaborative Analysis and Visualization promises to improve the way we carry out analysis by providing real-time support for visualization of spatial features and quantities”.

**Research Themes:** Web3D, Object-Oriented Computer Graphics, Collaborative Analysis and Visualization, Client Spatial Interpreter, Graphical Middleware and Software Engineering.

**European Scientific Computing Organization**

An ESCO proposal [137] was recently submitted to the 7th European Framework Program (FP7) which will encompass scientific visualization research work. ESCO will not cover all disciplines of physics (fluid dynamics, crash, biology, geophysics, chemistry…) nor all application domains (space, health, security, environment, energy…) but will focus on cross-disciplinary and cross-domain issues. Fluid dynamics will be coupled to crash simulations; Transport and Environment questions will be tackled together, etc. Fields of improvement have been identified around 4 themes:

- **Integrated Modeling Environments (IMEs):** Scientists largely use commercial, homemade or open-source Integrated Modeling Environments (Matlab, Scilab, Tent, Salome…). Providing new functionalities to IMEs and creating critical masses of users and professional services through technology providers is a key to success for the European scientific computing software industry.

- **Scientific visualization:** 2D/3D data acquisition and treatment are day-to-day business for many scientists. Yet there is still a lack of local computer resources for the pre- or post-processing of large data sets, a situation that could be solved by remote interaction. Collaboration with other scientists to compare models or results (a significant part of scientists’ job) could also be made easier.

- **Scientific software-packaging tools:** To ensure that an application reaches a critical mass of users, it is mandatory for the developer to build and package it on a multi-platform basis. Preparing packaging and testing environments on several operating systems (Linux, Windows, UNIX, MacOS…) is competence- and resource-consuming. Effective tools for packaging should be provided to developers.

- **Service-oriented interoperability middleware components:** To improve the interoperability of scientific software, it is proposed to develop an open set of middleware components to allow easy and standardized interfacing and access to modeling/computational tools, data exchanges, control commands definition, locally and remotely. Hence, the aim is to offer an open and extensive system of inter-compliant scientific tools (simulation codes and data processing) and make them easily available and useable locally and through the Web. This effort should help the users reap the benefits of state-of-art software and of up-to-date middleware. One technical solution to be explored is the Service-Oriented Approach (SOA). The SOA concept allows the development of composite software solutions, adaptable to tailored applications and easily customizable. The possibility of services ‘orchestration’ helps to identify modeling processes and favors the access to advanced tools by non-specialists.

**Development trends in Interactive Visualization systems**

The intensive interchange of design information and of test results and the distribution of tasks between the aerospace industry, the universities and the research institutes should be enabled on an international basis over a network like the Internet. Mechanisms for data encryption and security must be built into the configuration of the
The goal is to allow several/many users to simultaneously visualize information from various data sources on a large display wall, in multiple high-resolution 3D windows, in a situation that permits collaborative decision-making. The emerging "augmented reality systems" provide several advantages over conventional desktops [138]. Virtual reality environments may provide features such as true stereoscopy, 3D interaction, and individual/customized viewpoints for multiple users, enabling complete ‘natural’ collaboration at an affordable cost.

**Visualization and Simulation**

Visualization includes functionalities such as: accurate representation of a 3D model, its decomposition in components and realistic simulation of its physical behavior. Visualization differs from simulation in the sense that, while both are dynamic (and seen in animation sequences), simulation implies real-time computation with some amount of user control over the dynamics of the model (settings of parameters). In order to create an object-oriented framework for supporting the dynamic virtual collaboration, an interactive platform could be constructed with virtual worlds using specialized 3D objects that contain their own interaction information. A rigid body simulator can be used to calculate actor and object movements. If all object interactions are application-independent, a single scheme only is required to handle all interactions in the virtual world. Inverse kinematics could be used to increase the interaction possibilities and realism in collaborative virtual environments. This would result in a higher feeling of presence for connected users and allow for the easy, on-the-fly creation of new interactions. In order to allow interactivity the network load must be kept as low as possible.
Collaboration and User Interaction

The expected benefits of Collaborative Visualization (CV) are as follows [139, 140]: CV could:

- Enable collaborative decision-making
- Improve the insight into complex problems [141]
- Significantly reduce production and labor costs
- Streamline project analysis, design, engineering, and testing
- Eliminate the impracticality, hazard, or high-cost encountered in physical environments
- Demonstrate products, processes, and plans with highly-realistic simulations
- Enable the intuitive exploration and analysis of relationships between variables
- Allow interactive analyses of high-resolution, time-varying data sets (of theoretically unlimited size -- although limited in practice by available computational resources)

Collaboration functionalities include annotation, measuring, symbols and metaphors, version management and hierarchical classification, possibly through the use of ontology [142]. The implementation of new types of visualization techniques, such as disc trees, may overcome difficulties associated with the traditional representations of decision trees, such as visual clutter and occlusion by elements in the foreground (cone tree example) [143].

Without associating to the information the probability of certainty, analysis of the visualization would be incomplete, and could lead to inaccurate or incorrect conclusions. 3D reconfigurable disc trees could be used to provide users with the information visualization, together with uncertainty (e.g. expressed with different color attributes).

Human Machine Interface

An appropriate level of human-computer interaction needs to be implemented [144]. A first issue to be resolved concerns the level of intervention required in a particular context. The system would possibly provide a context-sensitive structure that enables users (otherwise unfamiliar with the system) to navigate through it. Three levels of intervention can be specified:

1. **Simplest**: to replicate a path the user is guided along a deterministic sequence of steps.
2. **More complex**: present alternatives to the user who selects from among them.
3. **Sophisticated**: to provide a critique of the process.

A second issue to address is configurability: different individuals, as well as different groups, have different views of a problem and its representation. One way to consider configurability is to specify a set of generic operations that would encompass the types of operations, information access and user tasks that need to be
supported by the system. If this list is compiled it can be structured hierarchically to provide several “depths” of intervention. This presumes that an analysis of the tasks required to accomplish goals has been conducted. A third issue lies with the need to handle different user roles -- such as “facilitator” or “mediator”; the system could be designed to support and facilitate the tasks of various roles. One of the typical participant’s roles, for example, is to understand a problem so that it can be defined and alternatives solutions generated and evaluated.

Input can be provided by (remote) sensors and other special equipment. The system could use sensor models and 3D scenic models to integrate video and image data from different sources [145]. Dynamic multi-texture projections enable real-time updating and “painting” of scenes to reflect the latest scenic data. Dynamic controls, including viewpoint as well as image inclusion, blending, and projection parameters, would permit interactive, real-time visualization of events. Mobile devices (PDA) could be used, as an alternative to mouse and keyboard, as user input devices.

![Figure 209: Mobile Device Controlling Virtual Worlds](image)

![Figure 210: Mobile Application Over Internet](image)

![Figure 211: Alternative User Interaction Devices](image)

![Figure 212: Handheld Devices](image)

A large multi-tiled display wall, driven by a system for parallel rendering running on clusters of workstations (e.g. Chromium [146]) can adequately satisfy the requirements of an output device for an advance visualization system. Several examples are given in Figure 214 to Figure 221.

![Figure 213: New generation of miniature computers and multi touch-screen inputs](image)
References


[38] B. Stroustrup, "Why C++ is not just an object-oriented programming language," in Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum) Austin, Texas, United States: ACM Press, 1995.


### Appendixes

**Lookup table and its C++ implementation for the pentahedron cell**

<table>
<thead>
<tr>
<th>Label</th>
<th>Node Mask</th>
<th>nF</th>
<th>nN</th>
<th>Edges Intersected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 0 0 1 1 3</td>
<td>0-3-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0 1 0 1 1 3</td>
<td>0-1-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 1 1 1 1 4</td>
<td>1-4-3-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0 0 0 1 0 0 0 1 3</td>
<td>1-2-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0 0 0 0 1 0 1 1 4</td>
<td>0-3-5-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0 0 0 0 1 1 0 1 4</td>
<td>0-2-5-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0 0 0 0 1 1 1 1 3</td>
<td>3-5-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0 0 0 1 0 0 0 1 3</td>
<td>3-6-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0 0 1 0 0 0 1 1 4</td>
<td>0-6-8-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0 0 1 0 1 0 2 6</td>
<td>0-4-1, 3-6-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0 0 1 0 1 1 1 1 5</td>
<td>1-4-6-8-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0 0 1 1 0 0 1 6</td>
<td>1-2-3-6-8-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0 0 1 1 1 0 1 2 7</td>
<td>4-7-6, 0-3-5-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0 0 1 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0 0 1 1 1 1 1 1 4</td>
<td>4-6-8-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0 1 0 0 0 0 0 1 3</td>
<td>4-7-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0 1 0 0 0 0 1 2 6</td>
<td>0-3-2, 4-7-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0 1 0 0 1 0 1 4</td>
<td>0-1-7-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>0 1 0 1 0 0 1 1 4</td>
<td>1-0-3-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0 1 0 1 0 0 1 5</td>
<td>0-6-7-5-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>0 1 0 1 0 1 2 7</td>
<td>3-5-7-6, 0-4-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0 1 0 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>0 1 0 1 1 1 1 4</td>
<td>3-5-7-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0 1 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>0 1 1 1 0 1 1</td>
<td>0-4-7-8-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0 1 1 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>0 1 1 1 1 1 1 4</td>
<td>1-7-8-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>0 1 1 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>0 1 1 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>0 1 1 1 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>0 1 1 1 1 1 1 3</td>
<td>5-7-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1 0 0 0 0 0 0 1 3</td>
<td>5-8-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>1 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>1 0 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>1 0 0 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1 0 0 0 1 0 0 1 4</td>
<td>1-2-8-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>1 0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>1 0 0 0 1 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>1 0 0 1 1 1 1 4</td>
<td>3-8-7-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1 0 1 0 0 0 1 4</td>
<td>3-6-7-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>1 0 1 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>1 0 1 0 1 0 1 7</td>
<td>0-1-4-6-7-5-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>1 0 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>1 0 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>1 0 1 1 0 1 1 4</td>
<td>0-6-7-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>1 0 1 1 0 1 2 6</td>
<td>0-2-3, 4-6-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>1 0 1 1 1 1 1 3</td>
<td>4-6-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>1 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>1 1 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1 1 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>1 1 0 0 0 1 1 2 6</td>
<td>1-5-2, 3-8-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>1 1 0 0 1 0 0 1 5</td>
<td>1-2-8-6-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>1 1 0 1 0 1 2 6</td>
<td>0-1-4, 3-8-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>1 1 0 1 1 0 1 4</td>
<td>0-2-8-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>1 1 0 1 1 1 1 3</td>
<td>3-8-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>1 1 1 0 0 0 1 3</td>
<td>3-4-5-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td>1 1 1 0 0 1 1 4</td>
<td>0-4-5-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>1 1 1 0 1 0 1 4</td>
<td>0-1-5-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>1 1 1 0 1 1 1 3</td>
<td>1-5-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>1 1 1 1 0 0 1 4</td>
<td>0-2-3-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>1 1 1 1 1 0 1 3</td>
<td>0-4-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>1 1 1 1 1 1 0 1 3</td>
<td>0-2-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>1 1 1 1 1 1 1 0 0</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 34: The lookup table for the pentahedron
---------------------------------------------------------------------------
/*                             CLASS Cell DEFINITION                         */
---------------------------------------------------------------------------
/*               V U B                                                       */
/*    Department of Fluid Mechanics                                 Oct 1993 */
/*           Dean Vucinic                                                    */
---------------------------------------------------------------------------
/*                                HEADER FILES                               */
---------------------------------------------------------------------------
// ----------------------- Pentahedron: -------------------------------
// nodes(edge)
UCharVec Cell3N6::NodesE_[9]={UCharVec("[0 1]"),
UCharVec("[1 2]"),
UCharVec("[2 0]"),
UCharVec("[0 3]"),
UCharVec("[1 4]"),
UCharVec("[2 5]"),
UCharVec("[3 4]"),
UCharVec("[4 5]"),
UCharVec("[5 3]");

// nodes(face)
UCharVec Cell3N6::NodesF_[5]={UCharVec("[0 2 1]"),
UCharVec("[2 0 3 5]"),
UCharVec("[0 1 4 3]"),
UCharVec("[1 2 5 4]"),
UCharVec("[3 4 5]"),

const Cell3N6::TP2T4 Cell3N6::X_[64]=
{
  0,0,((0,[-1,-1,-1,-1,-1,-1],[1,-1,-1,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  0,(([-1,-1,-1],[1,-1,-1],[-1,-1,-1]),
   ([-1,-1,-1],[-1,-1],[-1,-1,-1])),
  1,1,((3, [0, 3, 2,-1,-1,-1],[1,-1,-1,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  1,(([-1,-1,-1],[1,-1,-1],[-1,-1,-1]),
   ([-1,-1,-1],[-1,-1],[-1,-1,-1])),
  2,1,((3, [0, 1, 4,-1,-1,-1],[1,-1,-1,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  1,((0, 1, 4), [0, 3, 2],([-1,-1,-1],[-1,-1,-1]),
   ([-1,-1,-1],[1,-1,-1],[-1,-1,-1])),
  3,1,((4, [1, 4, 3, 2,-1,-1],[1,-1,-1,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  2,((1, 4, 3), [3, 2, 6],([1, 3, 2],[5, 1, 0]),
   ([-1,-1,-1],[1,-1,-1],[-1,-1,-1])),
  4,1,((3, [1, 2, 5,-1,-1,-1],[1,-1,-1,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  1,((1, 2, 5), [0, 3, 1],([-1,-1,-1],[-1,-1,-1]),
   ([-1,-1,-1],[1,-1,-1],[-1,-1,-1])),
  5,1,((4, [0, 3, 5, 1,-1,-1],[1,-1,-1,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  2,((0, 3, 5), [2, 1, 6],([0, 5, 1],[5, 3, 0]),
   ([-1,-1,-1],[1,-1,-1],[-1,-1,-1])),
  6,1,((4, [0, 2, 5, 4,-1,-1],[1,-1,-1,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  2,((0, 2, 5), [0, 1, 6],([0, 5, 4],[5, 3, 2]),
   ([-1,-1,-1],[1,-1,-1],[-1,-1,-1])),
  7,1,((3, [3, 5, 4,-1,-1,-1],[1, 3, 2,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  1,((3, 5, 4), [1, 3, 2],([-1,-1,-1],[-1,-1,-1]),
   ([-1,-1,-1],[1,-1,-1],[-1,-1,-1])),
  8,1,((3, [3, 6, 8,-1,-1,-1],[1, 3, 2,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  1,((3, 6, 8), [2, 4, 1],([-1,-1,-1],[-1,-1,-1]),
   ([-1,-1,-1],[1,-1,-1],[-1,-1,-1])),
  9,1,((4, [0, 6, 8, 2,-1,-1],[2, 4, 1, 0,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  2,((0, 6, 8), [2, 4, 6],([0, 8, 2],[5, 1, 0]),
   ([-1,-1,-1],[-1,-1,-1],[-1,-1,-1])),
 10,1,((6, [0, 1, 4, 6, 8, 3],[0, 3, 2,-1,-1,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
  4,((0, 1, 4), [0, 3, 6],([0, 4, 6],[5, 2, 7]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1])),
 11,1,((5, [1, 4, 6, 8,-2,-1],[3, 2, 4, 1, 0,-1]),
   (0,[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1]),
  3,((1, 4, 6), [3, 2, 6],([1, 6, 8],[5, 4, 7]),
   (1, 8, 2),[6, 1, 0],[[-1,-1,-1],[-1,-1,-1]]),
// nodes(facet)
UCharVec Cell3N6::NodesF_[5]={UCharVec("[0 2 1]"),
UCharVec("[2 0 3 5]"),
UCharVec("[0 1 4 3]"),
UCharVec("[1 2 5 4]"),
UCharVec("[3 4 5]"),

const Cell3N6::TP2T4 Cell3N6::X_[64]=
{
1. \{\{7, 8, 5\}, \{4, 1, 3\}, \{2, 3, 0\}, \{0, 1, 7\}\},
   \{\{0, 1, 7\}, \{2, 3, 0\}, \{4, 1, 3\}, \{7, 8, 5\}\},

32. \{\{3, 5, 8, 7, 1, -1\}, \{1, 4, 3, -1\}, \{6, 3, 2\}, \{5, 4, 7\}\},
   \{\{6, 3, 2\}, \{5, 4, 7\}, \{1, 4, 3\}, \{3, 5, 8, 7, 1, -1\}\},

33. \{\{6, 0, 3, 8, 7, 5, 2\}, \{2, 1, 4, 3\}, \{0, 1, 7\}, \{0, 1, 7\}\},
   \{\{0, 1, 7\}, \{0, 1, 7\}, \{2, 1, 4, 3\}, \{6, 0, 3, 8, 7, 5, 2\}\},

34. \{\{6, 0, 1, 5, 8, 7, 4\}, \{0, 3, 1, 4, 3, -1\}, \{0, 1, 5\}, \{0, 1, 5\}\},
   \{\{0, 1, 5\}, \{0, 1, 5\}, \{0, 3, 1, 4, 3, -1\}, \{6, 0, 1, 5, 8, 7, 4\}\},

35. \{\{3, 1, 5, 2, -1\}, \{3, 1, 0, -1\}, \{4, 2, 3, -1\}, \{3, 1, 5, 2, -1\}\},
   \{\{3, 1, 5, 2, -1\}, \{3, 1, 0, -1\}, \{4, 2, 3, -1\}, \{3, 1, 5, 2, -1\}\},

36. \{\{4, 1, 2, 8, 7, -1\}, \{0, 1, 4, 3, -1\}, \{0, 1, 4, 3, -1\}, \{0, 1, 4, 3, -1\}\},
   \{\{0, 1, 4, 3, -1\}, \{0, 1, 4, 3, -1\}, \{4, 1, 2, 8, 7, -1\}, \{0, 1, 4, 3, -1\}\},

37. \{\{5, 0, 3, 8, 7, 1\}, \{2, 1, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 1, 4, 3, 0\}, \{5, 0, 3, 8, 7, 1\}\},

38. \{\{5, 0, 2, 8, 7, 4, 1\}, \{0, 1, 3, 6\}, \{0, 2, 8\}, \{0, 2, 8\}\},
   \{\{0, 2, 8\}, \{0, 2, 8\}, \{0, 1, 3, 6\}, \{5, 0, 2, 8, 7, 4, 1\}\},

39. \{\{4, 3, 6, 7, 4, -1\}, \{1, 4, 3, -1\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{1, 4, 3, -1\}, \{4, 3, 6, 7, 4, -1\}\},

40. \{\{4, 3, 6, 7, 5, -1\}, \{2, 4, 3, -1\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, -1\}, \{4, 3, 6, 7, 5, -1\}\},

41. \{\{5, 0, 6, 7, 5, 2\}, \{2, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, 0\}, \{5, 0, 6, 7, 5, 2\}\},

42. \{\{4, 0, 1, 5, 3, -1\}, \{0, 3, 1, 2\}, \{3, 4, 6, 7, 1, -1\}, \{3, 4, 6, 7, 1, -1\}\},
   \{\{3, 4, 6, 7, 1, -1\}, \{3, 4, 6, 7, 1, -1\}, \{0, 3, 1, 2\}, \{4, 0, 1, 5, 3, -1\}\},

43. \{\{3, 1, 5, 2, -1\}, \{3, 1, 0, -1\}, \{3, 1, 0, -1\}, \{3, 1, 5, 2, -1\}\},
   \{\{3, 1, 0, -1\}, \{3, 1, 0, -1\}, \{3, 1, 5, 2, -1\}, \{3, 1, 5, 2, -1\}\},

44. \{\{3, 1, 5, 2, -1\}, \{3, 1, 0, -1\}, \{4, 2, 3\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{4, 2, 3\}, \{3, 1, 5, 2, -1\}, \{3, 1, 0, -1\}\},

45. \{\{4, 0, 6, 7, 1, -1\}, \{2, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, 0\}, \{4, 0, 6, 7, 1, -1\}\},

46. \{\{4, 0, 6, 7, 2, -1\}, \{2, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, 0\}, \{4, 0, 6, 7, 2, -1\}\},

47. \{\{4, 0, 6, 7, 2, -1\}, \{2, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, 0\}, \{4, 0, 6, 7, 2, -1\}\},

48. \{\{4, 0, 6, 7, 2, -1\}, \{2, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, 0\}, \{4, 0, 6, 7, 2, -1\}\},

49. \{\{4, 0, 6, 7, 2, -1\}, \{2, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, 0\}, \{4, 0, 6, 7, 2, -1\}\},

50. \{\{4, 5, 8, 6, -1\}, \{2, 4, 3, 0\}, \{0, 0, 3, 8\}, \{0, 0, 3, 8\}\},
   \{\{0, 0, 3, 8\}, \{0, 0, 3, 8\}, \{2, 4, 3, 0\}, \{4, 5, 8, 6, -1\}\},

237
[51, 2, {{3, {1, 5, 2, 3, 0, -1, -1, -1}},
    {3, 0, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[52, 1, {{0, 1, 0, 2, 1, 3, 4, 0},
    {0, 1, 0, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[53, 2, {{3, {0, 1, -1, -1, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[54, 1, {{3, {0, 1, 5, 2, 3, 0, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[55, 1, {{3, {0, 1, 5, 2, 3, 0, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[56, 1, {{3, {0, 1, 5, 2, 3, 0, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[57, 1, {{3, {0, 1, 5, 2, 3, 0, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[58, 0, {{0, 1, 5, 2, 3, 0, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[59, 1, {{3, {0, 1, 5, 2, 3, 0, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

[60, 1, {{3, {0, 1, 5, 2, 3, 0, -1, -1},
    {0, -1, -1, -1, -1, -1, -1, -1}},
    {0, 1, -1, -1, -1, -1, -1, -1}},
    {{-1, -1, -1, -1, -1, -1, -1, -1}},
    {1, 0, -1, 0, -1, -1, -1, -1}},
  ]

/*--------------------------------------------------------------------------*/
/*                         CLASS Cell END OF DEFINITION                      */
/*--------------------------------------------------------------------------*/
## Research Projects Timeline

<table>
<thead>
<tr>
<th>Year</th>
<th>Project</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990-1992</td>
<td>EUROVAL - EC/BRITÉ- EURAM project</td>
<td>An European initiative on validation of CFD codes</td>
</tr>
<tr>
<td>1993-1995</td>
<td>ECARP - EEC/BRITÉ- EURAM project</td>
<td>Validation of CFD models</td>
</tr>
<tr>
<td>1995-1999</td>
<td>CFDice - FWO project</td>
<td>An Integrated Computational Environment for CFD</td>
</tr>
<tr>
<td>1998-2001</td>
<td>ALICE – ESPRIT project</td>
<td>QFView - Quantitative Flow Field Visualization</td>
</tr>
<tr>
<td>2000-2004</td>
<td>QNET-CFD – FP5 GROWTH project</td>
<td>A thematic network for quality and trust in the industrial application of Computational Fluid Dynamics</td>
</tr>
<tr>
<td>2004-2005</td>
<td>LASCOT – EUREKA ITEA project</td>
<td>Large Scale Collaborative Decision Support Technology</td>
</tr>
<tr>
<td>2004-2007</td>
<td>NUSIC EU Tempus project</td>
<td>Numerical Simulation Curricula</td>
</tr>
<tr>
<td>2005-2008</td>
<td>NSP-ME EU Tempus project</td>
<td>Numerical Simulation Program in Mechanical Engineering</td>
</tr>
<tr>
<td>2006-2007</td>
<td>SERKET - EUREKA ITEA project</td>
<td>Security Keeps Threats away</td>
</tr>
</tbody>
</table>

*Table 35: Research Projects Timeline*
## Author’s publications at VUB

### 1989


### 1991


### 1992


### 1993


### 1994


### 1995


### 2000


Vucinic D., PIV measurements and CFD Computations of the double annular confined jet experiment , Pivnet T5/ERCOFTAC SIG 32 2nd Workshop on PIV, Lisbon July 7-8 2000.


Charles Hirsch, Dean Vucinic, editors of the QNET-CFD Newsletter:


Charles Hirsch, Dean Vucinic, editors of the QNET-CFD Newsletter:


### Research Developments Timeline

The development time of this thesis spans almost two decades of the author’s research and development activities, started when he joined VUB in September 1988. Somehow, these two decades naturally split his work in two distinctive periods:

1988-1997 is the period when the scientific visualization (SV) system was created, and further evolved to the industrial application

1998-2007 is the period when different parts of the developed methodology and software where applied to the IWT and EC projects

#### 1988-1997 period

1988

The SV system software development was initialized with a very immature C++ programming language pre-processor outputting C code, which needed to be further compiled in order to be executed. The available hardware was APOLLO workstations running a proprietary PHIGS graphic library (completely vendor-dependent environment). The initialization of the object-oriented software design was done for 2D structured and unstructured mono block data model.

1989

In the summer of 1989 the first implementation of the CFView software was accomplished, which was presented in September at the VKI Computer Graphics Lecture Series [40]. To the author’s knowledge, it was the first time ever object-oriented interactive SV software for fluid flow analysis had been developed. It is obvious that more powerful implementations existed, but these were not implemented with an object-oriented approach. That summer, Prof. Hirsch made a firm decision of allowing the author to continue working on his object-oriented methodology for developing CFView.

1990

As OOM was starting to gain acceptance, the C++ compiler came out. It immediately showed that C++ performance was starting to be comparable to C and FORTRAN. The CFView architecture had become more elaborated; the GUI and Graphics categories of classes had been modeled to enable the upgrade of CFView to the X-Windows C++ InterViews library from Stanford [147] and to the vendor-independent graphics library Figaro/PHIGS. The 3D structured mono-block data model was developed.

1991

As mentioned in the Introduction, this year was crucial in the CFView development: the main visualization system architecture was defined and the applied methodology was presented at the VKI Computer Graphics Lecture Series [41]. The 3D structured data model was extended to the multi-block data sets. 1991 was also the year when we received the first Silicon Graphics workstation running a specialized graphics library. It was the first time that 3D user interaction was performed with an acceptable system response time.

1992

In January 1992, CFView was presented at the AIAA meeting [21]; to the author’s knowledge, it was the first-time ever OO-application created for interactive fluid flow analysis. CFView was compared with visualization systems from NASA [16], MIT [17] and DLR [18] which had been implemented in C and developed with structured programming methodologies. Another system, similar in design to CFView, was Visage [148], came
out later on in 1992 at the IEEE meeting; Visage was implemented in C and developed at General Electric by the
group of researchers which developed in the late 90’s the OO VTK library. In 1992, the CFView data model
was extended to unstructured meshes based on tetrahedrons only. An upgrade to the new version of
Figaro/PHIGS was done, which improved the CFView graphics performance.

1993

As the CFView application became more complex, the development of a data model, as presented in this thesis,
was conceived for supporting transparent interactivity with structured and unstructured data sets. In this year, the
‘marching cube’ algorithm was extended and became a ‘marching cell’ algorithm for the treatment of
unstructured meshes with heterogeneous cell types, enhanced with unambiguous topology for the resulting
extracted surfaces. The Interviews library was upgraded to version 3.1 in order to support the CFView Macro
capability. The Parallel CFView was under ongoing development.

1994

The 3D structured and unstructured multi-block data model was completed, including the particle trace algorithm
for all cell types. The Parallel CFView was released and presented. As there were portability
problems with Figaro/PHIGS, the HOOPS graphics library was selected as the new 3D graphics platform for
CFView.

1995

The development of the HOOPS graphics layer was ongoing and the Model-View-Controller design was applied
to replace the interactive parts of CFView. This process resulted in a cleaner implementation which was
then prepared for porting onto the emerging PC Windows platforms enhanced with 3D graphics hardware
running OpenGL. The PC software porting and upgrade were done later on in NUMECA.

1996

Under the LCLMS project, the symbolic calculator for CFView was developed and investigations were carried
out to find an appropriate GUI platform, which had to be PC-Windows/UNIX portable. The result was the
Tcl/Tk library, which was found appropriate; it is still today the GUI platform for CFView.

1997

The CFView multimedia learning material was prototyped in the IWT LCLMS projects; the author continued to
advocate the use of his development methodology in the EU R&D-project arena, which attracted interest in R&D
partners and succeeded in bringing new EU-funded projects to the VUB.

1998-2007 period

1998

The development of QFView in the EC “ALICE” project extended the author’s research towards applying the
World Wide Web concept for designing and building distributed, collaborative scientific environments. The CFView data model was used for the development of QFView, which enabled combined visualization of CFD and EFD data sets. As described in the Introduction, 3 test cases were performed using the developed approach. It is important to mention that it is in the same project that the first PIV measurement system was established at the VUB (a premiere in the Belgian universities). This influenced the Department’s research to the integrated application of CFD and EFD in fluid flow analysis; today, the PIV measurements at VUB are performed with pulse laser.
QNET-CFD was a Thematic Network on Quality and Trust for the industrial applications of CFD [134]. The author’s contributions were more of a coordination and management nature, as he was entrusted with presenting and reporting on project activity, as well as preparing the publishing and dissemination material for the fluid flow knowledge base, involving web-site software development and maintenance.

2004

The LASCOT project [55] was about applying the Model-View-Controller (MVC) paradigm to enhance the interactivity of our 3D software components for: visualizing, monitoring and exchanging dynamic information, including space- and time-dependent data. The software development included the integration and customization of different visualization components based on 3D Computer Graphics (Java3D) and Web (X3D, SOAP) technologies, and applying the object-oriented approach based on Xj3D to improve decision-making situational awareness [153].

2006

The SERKET project -- currently in progress -- focuses on the development of ‘more-realistic’ X3D models for information visualization of security applications.

**Summarizing the Current State-of-the-Art**

Today, *scientific visualization* constitutes an important research area in Information Technologies (IT) and also in Engineering Sciences, with its main objective of representing data in a variety of pertinent visual forms. *SV* is here to help scientists and analysts, to better understand the results of their research, and to allow them to effectively convey these results to others by means of graphics, pictures and other visual ways (see also our Introduction, Section “SV Software - state of the art” and our final chapter “Conclusion and Future Development”).

The current *SV* systems do not only provide visualization tools: they also support the interfacing with complex computational codes, which facilitates the integration with the computing demands of expert users [10].

Interactive Visualization seems to continuously gain research interest [154], as there is a need to empower users with tools for extracting and visualizing important patterns in very large data sets. Unfortunately, for many application domains, it is not yet clear which are the features of interest, how to define them, let alone how they can be detected. There is a continuous need to develop new ways, which enable more intuitive user interactions, as a crucial element for further enhancement and exploration work. The interactive visualization tools for fluid flow analysis were discussed in Chapter 2 Adaptation of Visualization Tools; no doubt that ongoing research will lead to the expansion and the improvement of the capabilities of the tools that were developed in the context of the author’s work.

Several current visualization development frameworks focus on providing customized graphical components for high-end desktops [155] and compatible with Ajax [156] and Eclipse [157] Open Source development environments. These components are based on the MVC paradigm; they deliver point-and-click interaction tools based on specialized SDKs, which allow the design of intuitive graphical displays for new applications. As discussed in Chapter 3, “Object-Oriented Software Development” will remain an integral part of tomorrow’s software development process.

Animation has become an important element of modern *SV* systems, and new video coding technologies will need to be taken into account in future developments. An example of this new technology is *MESHGRID*, developed at the VUB/ETRO [158-160] and adopted by MPEG-4 AFX [161]. The model-based representation provided by the *MESHGRID* technology could be used for modeling time-dependent surfaces (in association with
the iso-surface algorithm discussed in this thesis) including video compression. This representation combines a regular 3D grid of points, called the reference-grid, with the wire-frame model of a surface. The extended functionality of MESHGRID could provide a hierarchical, multi-resolution structure for animation purposes, which allows the highly compact coding of mesh data; it could be considered for integration in a new generation of SV software for fluid flow time-dependent analysis. Other MESHGRID elements that could be considered for SV software are view dependency (not considered in this thesis) and Region of Interest (ROI) -- a concept which can be associated to the multi-block (multi-domain) model of fluid flow data; it offers the possibility of restricting visualization to a limited part of the whole data set. The MESHGRID data model seems appropriate for implementation in distributed environments since it provides advanced data compression and data transfer techniques that are important for the quality of service in animation enhanced software.

Concerns of performance are at the core of the development of VS systems distributed over desktops. Such systems are capable of very large-scale parallel computation and of distributed rendering on large display walls or ‘immersive’ virtual environments. Today, modern graphics hardware (Graphics Processing Units -- GPU) performs complex arithmetic at increasingly high speed. It can be envisaged that GPU-s can be used to execute SV non-graphics algorithms, which offers a potential for increasing VS performance for large and complex data sets without sacrificing interactivity.

Current trends in SV research and development help advance the state-of-the-art of computational science and engineering by:

- investigating high performance computing;
- exploring collaborative approaches that encompass multi-disciplinary, team-oriented concepts
- applying distributed software with visualization and virtual reality components that improve the ergonomics of human-machine interfaces.

These aspects are further covered in the concluding Chapter “Development trends in Interactive Visualization systems”.

The 2007 “Advanced Scientific Computing Research” program of the US Department of Energy [162] includes SV research and development work in relation to advances in computer hardware (such as high-speed disk storage systems, archival data storage systems) and high-performance visualization hardware. An example of a high-performance computer network is the UltraScienceNet Network (USNET) Testbed, a 20 gigabit-per-second, highly-reconfigurable optical network that supports petabyte data transfer, remote computational steering and collaborative high-end visualization. USNET provides capacities that range from 50 megabits-per-second to 20 gigabits-per-second. Such capability is in complete contrast with the Internet where the shared connections are provided statically with a resulting bandwidth that is neither guaranteed nor stable.

Ongoing research in visualization tools for scientific simulation is exploring hardware configurations with thousands of processors and developing data management software capable of handling terabyte-large data sets extracted from petabyte-large data archives. It is predicted that large-scale, distributed, real-time scientific visualization and collaboration tools will provide new ways of designing and carrying out scientific work, with distributed R&D teams in geographically distant institutes, universities and industries accessing and sharing in real-time extremely powerful computational and knowledge resources, and yet to be measured gains in efficiency and productivity.
Performance Analysis

In order to assess the capability of the CFView system, a performance analysis was conducted in the EC PASHA project [127, 150]. The main goal was to see how two parallel ‘SIMD’ and ‘MIMD’ implementations of CFView would perform compared to its sequential implementation on a ‘SISD’ stand-alone machine. The benchmarks consisted of test cases specifically designed to provide a meaningful, reliable basis for comparing the performances of the implementations. The test cases were theoretically-minded, not necessarily representative of practical applications. These test cases were retained because they offered significant advantages over real-world examples, as follows:

1. The test cases were not biased towards any particular algorithms, computer systems or applications. This property makes them good candidates for comparisons between different systems.
2. The test cases were based on simple, algorithmic definitions. This made them readily available to anyone wanting to perform similar experiments.
3. The test cases were all different, so as to avoid the testing processes to focus upon the singularities of one particular data set.
4. Single test cases were devised to explore a wide spectrum of difficult, uncommon, challenging or otherwise interesting characteristics. This enabled us to considerably speed up the testing, debugging and optimization procedures.
5. The test cases were designed to offer the user the possibility to control the complexity of the problem (e.g. control over the size and complexity of iso-surfaces).

The test cases that were developed and used are described below. An overview of the testing environment is then given, including a description of the hardware used, of the algorithms tested and of the measurements taken. The results of the experiments are reported and summarized in tables. A discussion of the results is provided showing the performance and characteristics of the systems and algorithms under test. Finally, some conclusions are presented.

Theoretical Test Cases

Specifically designed test cases have been used to benchmark the distributed, SIMD and MIMD CFView implementations. For each test case, one ‘low-‘ and one ‘medium-‘ volume cases were considered. Specifically, for the cutting plane and the iso-surface experiments, one used a geometrical data set (the mesh) and a scalar quantity data set (the scalar field). The dimensions of the low- and medium-volume cases were respectively of 20*20*20 (8000 vertices) and 50*50*50 (125000 vertices). Size-wise, the full test cases -- the mesh plus the scalar field -- add up to some 128K (low) and 2MB (medium) of data.

The data were generated by the following piece of C-code:

```c
extern random() /* returns random number: 0 <= random() < 1 */ {
    FILE *geo_file, *scal_file;
    int i, j, k;
    int max_i, max_j, max_k;
    /* data for the low data volume case: 20*20*20 vertices */
    max_i = 20; /* number of vertices in dimension i */
    max_j = 20; /* number of vertices in dimension j */
    max_k = 20; /* number of vertices in dimension k */
    /* open the data files */
    geo_file = fopen("low-20.geo", "w");
    scal_file = fopen("low-20.sca", "w");
```
/* random data generation */

for (k=0; k<max_k; k++){
    for (j=0; j<max_j; j++){
        for (i=0; i<max_i; i++){
            fprintf( geo_file, "%f %f %f\n",
                     random()+(float) i,
                     random()+(float) j,
                     random()+(float) k );
            fprintf( scal_file, "%f\n", 2 * random() - 1 );
        }
    }
}

This code generates structured meshes with a random distortion and scalar fields with values that randomly vary between -1 and 1. The mesh vertices coordinates are the \((i,j,k)\) indices of the corresponding mesh cell. Each coordinate is incremented with a positive random number strictly smaller than 1, which leads to the definition of a structured mesh with strongly deformed cells.

The particle-tracing benchmarks were run with a helical vector-field data set defined over a regular (not distorted), structured mesh. Here also, a low- and a medium-volume test cases were considered, respectively with 10*10*10 (1000 vertices) and 30*30*30 (= 27000 vertices). The helical field \((u,v,w)\) was computed for every vertex \((x,y,z)\) using the spiral equation:

\[
(u,v,w) = (ax-by, bx+ay, -2az+c)
\]

with parameters set to: \(a=0; b=1; c=1\). The full test cases (mesh and helical field) accounted for some 25K (low) and 650K (medium) of data.

**Hardware Environment**

**SISD computer:**
- HP 9000/735 (under HP-UX 9.0.1)
- 99 MHz clock speed
- 96 MB RAM
- CRX-24Z graphics board
- 124 MIPS peak performance
- 40 MFLOPS peak performance

**SIMD computer:**
- CPP DAP 510C-16
- 1024 bit-processors each with a 16-bit floating point coprocessor
- 16 MB RAM (shared)
- 140 MFLOPS peak performance
- SCSI connection with front end

**MIMD computer:**
- Parsytec GC-1/32 (under Parix)
- 32 processors (Inmos T8, RISC)
- 128 MB RAM (4 MB local memory per processor)
- 140 MFLOPS peak performance
- S-bus connection with front end.

To be able to visualize the results of the computations, the parallel computers were linked to a front-end workstation running the CFView interface. This work-station was the computer separately used for the SISD benchmarks (see above). The SISD machine was connected to a Parallel Server via an Ethernet LAN (10 Mbits/s). The Parallel Server was running the largest part of the Interface Framework, built on top of *PVM*, see Figure 194. It communicates with the SIMD and MIMD computers through its SCSI-bus and through its internal S-bus respectively. For the Parallel Server machine the SUN workstation with the following characteristics was used:
SUN SPARCstation 10/30GX
- SuperSPARC processor
- 32 MB RAM
- 86 MIPS peak performance
- 10 MFLOPS peak performance

A schematic view of the environment used for the benchmarking is shown in Figure 222.

![Figure 222: Overview of the heterogeneous and distributed environment used for the theoretical benchmarks](image)

**Algorithms Parameters**

The performance tests were done for 3 extraction algorithms: cutting-plane, iso-surface and particle-tracing. For each algorithm, the tests were run with parameter ranges chosen to obtain data from very low computational load (e.g. a plane that intersects only with a very small part of the mesh) to very high load (e.g. a plane that intersects a very large part of the mesh).

For the cutting-plane algorithm, the intersection plane is defined by the normal (1,1,1) and the intersection with the X-axis linearly varying between 0 and 2 times the x-dim of the mesh. Given the characteristics of the meshes used, this means that the largest intersection will be found at x=1.5*x-dim.

For the iso-surface computations, a range of iso-values was chosen to generate an (approximately) linearly-varying number of triangles (out of which the iso-surface is built) keeping their number in the range, which the computers could handle.

For the particle-tracing benchmarks, the number of particles launched (at some fixed time t) was varied. In a parallel configuration, multiple vector-field lines may be computed at the same time from a single data set: this is achieved, essentially, by allowing the parallel processors to work simultaneously on different field lines. For the CFView user, launching many particles at the same time (usually by distributing them evenly along a line segment) is a common procedure. Typically, the user first looks for interesting regions of the model by studying the traces of individually-placed particles. Then, the user positions a suite of particles in a region of interest.
Being located fairly close to each other, the computed particles traces form a ‘ribbon’ whose motion shows the particularities of the flow field. Clearly, one expects that the (speed-up) effects of parallel computation to be comparatively stronger at high particle numbers. By varying the number of particles from a few to very many, one was able to explore the effects of parallelism with respect to sequential computation.

Figure 223 (a) illustrates the benchmarking mesh data (20x20x20 vertices) used in the PASHA project. Figure 223 (b) shows the 10-million-point mesh (~300MB data on disk), which was created (on the very week of publication of this thesis) to demonstrate that CFView is capable of handling data sets of higher orders of magnitude. The 3 visualization algorithms were executed on such large data sets. Figure 224 (a) shows a cutting plane with particle traces illustrating the benchmarked helical vector field. Figure 224 (b) shows the iso-surface extracted from the complex- and unconventional-geometry test case. The apparently-perfect spherical shape of the iso-surface demonstrates the correctness of the extracted data. The ‘clean’ helical geometry of the particle traces is a visual indicator of the correctness of the algorithm applied here in trying conditions, namely to a vector field defined on a distorted mesh with 200x200x250 vertices. For the record, these computations were run on a Dell XPS-M2010 with Intel Dual Core 2 CPU @ 2.33 MHz and 2GB RAM.
Measurements Timing

When analyzing the performance of distributed computation, one must account for the (unavoidable) overhead due to the network background tasks and the multiple processes that govern the calculations (e.g. their synchronization). This is why time in the tests was measured using ‘wall-clock’ time as ‘CPU’ time was not available. The time figures used include overhead due to the activity of various UNIX processes and to the network load caused by system activities (swapping, saving, etc.).

Time was measured using the C function “ftime” with a resolution of about 1 millisecond. Measurements were made on the individual workstations involved. The network times were either inferred from the measured times (when possible) or computed through explicit “handshaking” between the communicating processes.

In order to obtain realistic performance figures, the measurements were performed in a ‘typical’ usage situation, i.e. with a ‘normal’ network load, ‘normal’ computer load, etc. In this approach, the part which includes overhead in the timing can be considered as ‘normal’ -- and indeed unavoidable for the user.

Because of the heterogeneous and distributed nature of the test implementation (SIMD/MIMD), timing a test run requires combining several measurements obtained on sub-parts of the system. Every single run of a particular algorithm was divided in a number of consecutive stages, and timing measured for each stage, namely:

- **PVMRcvGeo** is the time it takes for the geometry data to go over the LAN from CFView to the parallel server.
- **GCSndGeo** or **DAPSndGeo** is the time it takes for the geometry data to go from the parallel server to the parallel machines (Parsytec GC and DAP respectively).
- **PVMRcvScal** is the time it takes for the scalar quantity data to go over the LAN from CFView to the parallel server.
- **GCSndScal** or **DAPSndScal** is the time it takes for the scalar quantity data to travel from the parallel server to the parallel machines (Parsytec GC and DAP respectively).
- **PVMRcvParam** is the time it takes for the parameter values (equation of the plane, the particular iso-value, the initial positions of the particles, etc.) to travel over the LAN from CFView to the parallel server.
- **GCExe** or **DAPExe** is the time it takes for the respective parallel machines to execute the given algorithm using the data stored in their own memory.
- **GCRcvResults** or **DAPRcvResults** is the time it takes for the respective parallel machines to return the results of their computation (collections of triangles or streamlines) to the parallel server.
- **PVMSndResults** is the time it takes for the computation results to travel over the LAN, from the parallel server to CFView (running on a workstation).
- **Render** is the time it takes for CFView to render the results on the screen of the workstation it is running on.

Measurements Results

We present below the characteristic results of the benchmarks. A result table is given for each algorithm; it shows ‘averaged algorithm-execution times’ on the different systems for several test cases. Averages are computed over 20 runs for each test case. All time values in the Tables below are in seconds.

For the CFView SIMD and MIMD implementations, the following average time values are given:

- (Send Data): time needed to send the data (mesh and scalar quantity) from the Workstation to the Parallel Server;
• (Load Data). time needed for loading the data from the Parallel Server onto the parallel machines;

• (Execute): execution time on the parallel implementations, which includes:
  
  (i) sending of the algorithmic parameters from the Workstation to the Parallel Machines;
  
  (ii) execution of the algorithm on the parallel machines;
  
  (iii) sending the result data from the parallel machines to the Parallel Server; and
  
  (iv) sending the result data from the Parallel Server to the Workstation.

For the CFView SISD implementation, only the averaged execution time is meaningful and shown. The averaged number of triangles generated by the algorithm and the number of particles traced are given where relevant. All times shown are averaged over 20 different runs, with parameters varying (see “Algorithmic parameters” above).

The total execution time for the parallel implementations is the sum of the three timings: Send, Load, and Execute. However, by making use of a caching mechanism in the Interface Framework, the sending and the loading of the data needs to be done once only (in the beginning). Thereafter, data to be transmitted to the parallel machines include only a (new) cutting-plane equation, an iso-value, or a set of initial particle positions, after which the execution can take off. Hence, in a realistic situation, only the times in the (Execute) column are relevant for comparison.

### Table 36: Average times for Cutting Plane (wall-clock time in seconds)

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>#Triangles</th>
<th>System</th>
<th>Send Data</th>
<th>Load Data</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>915</td>
<td>SISD</td>
<td>----</td>
<td>----</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMD</td>
<td>0.26</td>
<td>0.48</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIMD</td>
<td>0.26</td>
<td>1.96</td>
<td>1.63</td>
</tr>
<tr>
<td>medium</td>
<td>10150</td>
<td>SISD</td>
<td>----</td>
<td>----</td>
<td>6.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMD</td>
<td>3.66</td>
<td>5.82</td>
<td>3.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIMD</td>
<td>3.66</td>
<td>28.86</td>
<td>4.44</td>
</tr>
</tbody>
</table>

### Table 37: Average times for Isosurface (wall-clock time in seconds)

<table>
<thead>
<tr>
<th>Mesh size</th>
<th>#Triangles</th>
<th>System</th>
<th>Send Data</th>
<th>Load Data</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>11145</td>
<td>SISD</td>
<td>----</td>
<td>----</td>
<td>9.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMD</td>
<td>0.26</td>
<td>0.48</td>
<td>6.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIMD</td>
<td>0.26</td>
<td>1.96</td>
<td>5.13</td>
</tr>
<tr>
<td>medium</td>
<td>10150</td>
<td>SISD</td>
<td>----</td>
<td>----</td>
<td>27.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMD</td>
<td>3.66</td>
<td>5.82</td>
<td>6.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIMD</td>
<td>3.66</td>
<td>28.86</td>
<td>4.96</td>
</tr>
</tbody>
</table>

### Table 38: Average times for Particle Trace (wall-clock time in seconds)

For the Particle Trace measurements the initial positions of the particles were evenly distributed along an imaginary line segment within the meshes. The particle trace input parameters were set as follows:

- average number of integrations per cell: 10
- maximum length of 1 particle path: 1000
- Forward and Backward tracing yes

The results of the experiments are summarized in Table 39. It shows the average execution times for the tracing of N particles, with N = 2, 5, 10, 15... 50.
Table 39: Evolution of the execution times in seconds with the number of particles used

<table>
<thead>
<tr>
<th>#processors</th>
<th>MIMD Calculate</th>
<th>MIMD Retrieve</th>
<th>MIMD Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.9</td>
<td>1.47</td>
<td>0.39</td>
</tr>
<tr>
<td>2</td>
<td>1.49</td>
<td>1.47</td>
<td>0.40</td>
</tr>
<tr>
<td>4</td>
<td>1.33</td>
<td>1.49</td>
<td>0.39</td>
</tr>
<tr>
<td>8</td>
<td>1.26</td>
<td>1.51</td>
<td>0.39</td>
</tr>
<tr>
<td>12</td>
<td>1.21</td>
<td>1.54</td>
<td>0.41</td>
</tr>
<tr>
<td>16</td>
<td>1.21</td>
<td>1.54</td>
<td>0.41</td>
</tr>
<tr>
<td>25</td>
<td>67.04</td>
<td>17.14</td>
<td>85.27</td>
</tr>
<tr>
<td>50</td>
<td>122.99</td>
<td>27.58</td>
<td>170.30</td>
</tr>
<tr>
<td>25</td>
<td>67.04</td>
<td>17.14</td>
<td>85.27</td>
</tr>
</tbody>
</table>

Table 39: Evolution of the execution times in seconds with the number of particles used

Table 40: Execution times in seconds for Isosurface on MIMD for different machine configurations (wall-clock time) with varying number of processors

Since the cutting plane and isosurface algorithms are conceptually very similar [151], comparing the performance results for them makes sense from a user’s point of view. The timing values as seen by the user are consolidated in Figure 225. The chart shows the averaged execution times for the cutting-plane and iso-surface algorithms on the different machines. For the SIMD and MIMD implementations, the execution times are subdivided in three parts:

(i) the actual algorithm-execution time on the given machine (Calculate);
(ii) the time it takes to transfer the computation result data from the parallel machine to the Parallel Server (Retrieve Results); and
(iii) the time needed to send the result data from the Parallel Server to Parallel CFView running on the Workstation (Send Results).

Figure 225: Average execution times in seconds for the algorithms on the different machines (with caching mechanism enabled for the parallel implementations).
Figure 226: Average execution times in seconds for the SIMD and MIMD implementations of the isosurface algorithm, with respect to the number of triangles generated (caching mechanism on)

As can be seen on the chart, the execution times for the parallel implementations are significantly shorter than for the sequential one. This is especially visible for the iso-surface algorithm because it requires a sweep through the complete mesh. Note that the machine running the sequential CFView was a state-of-the-art HP 9000/735 workstation, at the time amongst the fastest in its category. This demonstrated that the parallel CFView SIMD/MIMD implementations offered intrinsically more power than the sequential CFView.

A second remarkable observation related to the distribution of computational load on the parallel machines. Although both implementations are approximately equally fast, it was found that the MIMD machine spent most of its CPU time for transferring results data to the parallel server, whereas the SIMD machine was 'busier' with the actual computation. The reason for this is, of course, the overhead caused by having to route the data between the different processors of the MIMD machine. Figure 226 shows this difference more clearly.

As seen from the chart above, the total execution time (as seen by the user) varies with the number of triangles generated, that is, with the complexity of the computation. On the SIMD machine, the largest contributing factor is calculation -- the time for computing the iso-surface. On the MIMD implementation, however, calculation times (MIMD Calculate) are fairly independent of the number of triangles generated. The time required to move the results from the MIMD machine to the Parallel Server (MIMD Retrieve Results) is, by contrast, largely dependent on the number of triangles. As mentioned earlier, this behavior results from the need to route the results of the computations by the different processors to the ‘master’ processor (which governs the communications with the parallel server, which is a time-consuming process.

An interesting observation can be made concerning the difference in performance between the parallel machines. As long as the number of triangles generated is small, the SIMD machine outperforms the MIMD machine. But when the task requires more and more computation (more intersections need to be calculated and more triangles need to be constructed), the situation reverses. This phenomenon can be explained as follows: for implementing the iso-surface algorithm in a fully parallel mode, one requires the ability to index the data set in parallel, a feature which is not available on the SIMD machine. Therefore, part of the computation is implemented as a serial loop over the edges of the intersected mesh cells. As the number of intersections (hence the number of triangles) increases, the serial part of the algorithm tends to dominate the computation and slows down the overall execution. One can also see in the charts that the local networking tasks (SIMD/MIMD Send Results), which ensure the communication in the heterogeneous distributed environment, account for only a small fraction.
of the total time. This suggests that a heterogeneous, distributed approach is feasible since it does not impose unacceptable overhead on the overall performance of the system.

A comparison of the execution times for particle-tracing on the SIMD and SISD machines can be found in Figure 227. As one can infer from the chart, the execution times on the SIMD machine raise linearly with the number of particles traced. The performance of the SISD machine quickly degrades as the number of particles increases; mesh size does not seem to influence this behavior. The chart shows that the parallel implementation succeeds in keeping execution times at an acceptable level, even for very large number of particles.

The SIMD implementation seems to be able to fully exploit the computing power that results from the distribution of the particle-tracing calculations onto several independent parallel processors.

The benchmarking experiment on the SIMD and MIMD CFView systems has shown comparable performance for the two parallel machines. The execution times were found to be comparatively more sensitive to the amount of computation required on the SIMD machine, and relatively more dependent on the amount of data to be routed between the different processors on the MIMD implementation. The overhead induced by the Interface Framework (developed for communication between the different machines) was seen to contribute to a small fraction only of the overall execution times.

It was also observed that the SIMD implementation really takes advantage of its multi-processor structure for particle-tracing. Indeed, the SIMD machine was demonstrated to be the only useable machine for tracing moderately-large numbers of particles.

The overall performance of the SIMD and MIMD Parallel CFView implementations were shown to be significantly better than the performance of the SIMD sequential version, especially for computationally-intensive operations, such as iso-surface construction on problems with large data volumes or particle-tracing with large numbers of particles.

Overall, the benchmarking of CFView has demonstrated that heterogeneous and distributed SV system configurations were indeed a viable proposition. This opens the interesting prospect of transparently using VS systems capable of harnessing the computing power of different computing platforms and taking advantage of geographically-distant, parallel machines.

![Figure 227: Execution times in seconds for particle tracing with respect to the number of particles](image-url)
VUB Double Annular Jet Experiment in turbulent flow regime:

LDA (Laser Doppler Anemometry)

CFD with Baldwin-Lomax model

LSV (Laser Sheet Visualization) during transition

PIV (Particle Image Velocimetry) averaged from 100 frames